



兰州大学

# 课 程 论 文

(本科生)

论文题目（中文） 《数据结构》课程的学习报告

论文题目（英文） Study Report of Data Structures

学 生 姓 名 谭源

导师姓名、职称 蒙应杰

学生所属学院 信息科学与工程学院

专 业 计算机类

年 级 2019 级

兰州大学教务处

## 诚信责任书

本人郑重声明：本人所呈交的毕业论文（设计），是在导师的指导下独立进行研究所取得的成果。毕业论文（设计）中凡引用他人已经发表或未发表的成果、数据、观点等，均已明确注明出处。除文中已经注明引用的内容外，不包含任何其他个人或集体已经发表或在网上发表的论文。

本声明的法律责任由本人承担。

论文作者签名：\_\_\_\_\_ 日期：\_\_\_\_\_

## 关于毕业论文（设计）使用授权的声明

本人在导师指导下所完成的论文及相关的职务作品，知识产权归属兰州大学。本人完全了解兰州大学有关保存、使用毕业论文（设计）的规定，同意学校保存或向国家有关部门或机构送交论文的纸质版和电子版，允许论文被查阅和借阅；本人授权兰州大学可以将本毕业论文（设计）的全部或部分内容编入有关数据库进行检索，可以采用任何复制手段保存和汇编本毕业论文（设计）。本人离校后发表、使用毕业论文（设计）或与该毕业论文（设计）直接相关的学术论文或成果时，第一署名单位仍然为兰州大学。

本毕业论文（设计）研究内容：

- 可以公开
- 不宜公开，已在学位办公室办理保密申请，解密后适用本授权书。

（请在以上选项内选择其中一项打“√”）

论文作者签名：\_\_\_\_\_ 导师签名：\_\_\_\_\_

日期：\_\_\_\_\_ 日期：\_\_\_\_\_

# 《数据结构》课程的学习报告

## 中文摘要

在计算机科学中，数据结构是计算机中存储、组织数据的方式。这学期使用《数据结构》[1]，蒙老师先由数据结构的定义开始讲起，再依次讲解了算法、线性表、栈和队列、串、数组和广义表、树形结构、图结构、排序、数据检索这几部分内容。

本文使用 LaTeX 编写，并使用 git 管理项目文件。项目已被开源到 GitHub，老师可以通过查看 commit 记录来了解我学习的过程。项目地址为<https://github.com/Reset12138/LZU-Data-Structures>，点击此超链接即可打开。

因时间缘故该版本还存在许多疏漏，如有不妥之处同学们可以 Pull a Request。

**关键词：** 数据结构，综述，LaTeX

# **STUDY REPORT OF DATA STRUCTURES**

## **Abstract**

This essay explores the world of Data Structures. In computer science, a data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

**Key Words:** data structure; review; LaTeX.

# 目 录

中文摘要 .....	I
英文摘要 .....	II
第一章 数据结构绪论 .....	1
1.1 数据结构的基本概念及研究内容 .....	1
1.2 数据结构的选择与评价 .....	2
第二章 算法 .....	3
2.1 算法的定义 .....	3
2.1.1 概述 .....	3
2.1.2 定义 .....	3
2.1.3 内涵及分类 .....	3
2.1.4 算法与程序的异同 .....	3
2.2 算法的描述及设计原则 .....	3
2.2.1 算法描述方法 .....	3
2.2.2 算法基本标准 .....	4
2.3 算法分析概论及有效算法 .....	4
2.3.1 概念 .....	4
2.3.2 复杂性分析 .....	5
2.4 算法设计方法概论 .....	6
2.4.1 概述 .....	6
2.4.2 算法设计的基本技术 .....	6
2.5 算法描述语言 .....	7
2.5.1 PDL 概述 .....	7
2.5.2 PDL 的优势 .....	7
2.5.3 PDL 书写及要求 .....	7

第三章 线性表 .....	12
3.1 线性表及其运算 .....	12
3.1.1 线性表的定义 .....	12
3.1.2 线性表的特征 .....	12
3.1.3 线性表的运算 .....	12
3.2 线性表的储存表示 .....	12
第四章 栈和队列 .....	15
4.1 栈及其运算 .....	15
4.1.1 绪论 .....	15
4.1.2 栈的基本定义 .....	15
4.1.3 与线性表的关系 .....	15
4.1.4 栈的运算 .....	15
4.1.5 栈的存储与运算的实现 .....	15
4.1.6 多栈共存问题 .....	15
4.2 栈的应用 .....	15
4.3 队列及其运算 .....	15
4.4 受限的栈及队列（了解） .....	16
第五章 串 .....	17
5.1 串及其运算 .....	17
5.2 串的模式匹配 .....	18
5.3 例题 .....	19
第六章 数组和广义表 .....	23
6.1 数组的定义与运算 .....	23
6.1.1 定义 .....	23
6.1.2 运算 .....	23
6.2 数组元素的地址访问 .....	23
6.2.1 概述 .....	23
6.2.2 地址计算 .....	24

6.2.3 特殊数组 .....	25
6.3 稀疏矩阵 .....	26
6.3.1 概要 .....	26
6.3.2 定义 .....	26
6.3.3 稀疏矩阵的存储 .....	27
6.3.4 稀疏矩阵的运算 .....	29
6.4 广义表（了解） .....	29
6.4.1 定义 .....	29
6.4.2 特性 .....	30
6.4.3 存储 .....	30
<b>第七章 树形结构.....</b>	<b>31</b>
7.1 树的基本定义和运算 .....	31
7.1.1 树形结构的概述 .....	31
7.1.2 树的基本定义 .....	31
7.1.3 有序树的基本定义 .....	31
7.1.4 森林（树林） .....	32
7.2 二叉树 .....	32
7.3 遍历二叉树 .....	32
7.3.1 概念 .....	32
7.3.2 二叉树的遍历次序（order） .....	32
7.4 树、森林与二叉树的转换 .....	33
7.4.1 概述 .....	33
7.4.2 树转为二叉树 .....	33
7.4.3 二叉树还原为树 .....	34
7.4.4 森林转为二叉树 .....	34
7.4.5 二叉树转为森林 .....	35
7.4.6 树的遍历 .....	35
7.4.7 森林的遍历 .....	36
7.5 线索树 .....	37
7.6 树形结构的应用 .....	37
<b>第八章 图结构.....</b>	<b>38</b>

8.1	基本概念 .....	38
8.1.1	概述 .....	38
8.1.2	概念 .....	38
8.2	图的存储 .....	42
8.2.1	邻接矩阵 .....	42
8.2.2	邻接表 .....	42
8.2.3	邻接多重表 .....	44
8.3	图的遍历 .....	44
8.3.1	概述 .....	44
8.3.2	图的深度优先遍历 .....	45
8.3.3	图的广度优先遍历 .....	45
8.4	连通性及最小生成树 .....	45
8.4.1	连通分量和生成树 .....	45
8.4.2	最小生成树 .....	46
8.5	有向无环图及应用 .....	47
8.5.1	拓扑排序 .....	47
8.5.2	关键路径 .....	50
8.6	最短路径 .....	51
8.6.1	单源最短路径 .....	51
8.6.2	每对顶点的最短路径 (all-pairs shortest path) .....	51
	第九章 排序 .....	52

9.1	基本概念 .....	52
9.1.1	概论 .....	52
9.1.2	相关概念 .....	52
9.1.3	存储结构设计 .....	54
9.2	插入排序 .....	54
9.2.1	插入类排序基本方法 .....	54
9.2.2	直接插入排序 (straight insertion sort) .....	54
9.2.3	二分插入排序 (dichotomising insertion sort) .....	55
9.2.4	2-路插入排序 .....	56
9.2.5	表插入排序 (list inserting sort) .....	56

9.2.6 希尔排序 (Shell sort) .....	56
9.3 交换排序.....	57
9.3.1 交换类排序基本方法.....	57
9.3.2 起泡排序 (bubble sort).....	57
9.3.3 快速排序 (quick sort).....	58
9.4 选择排序.....	59
9.4.1 选择类排序基本方法.....	59
9.4.2 直接选择排序 (straight selection sort) .....	59
9.4.3 树形选择排序 (tree selection sort) .....	59
9.4.4 堆排序 (heap sort) .....	59
9.5 合并排序.....	59
9.5.1 合并类排序基本方法.....	59
9.5.2 两组合并 .....	59
9.5.3 二路归并排序.....	59
9.6 枚举排序.....	60
9.6.1 计数类排序基本方法.....	60
9.6.2 计数类排序的过程.....	60
9.7 分配排序.....	60
9.7.1 分配排序概述.....	60
<b>第十章 数据检索.....</b>	<b>61</b>
10.1 基本概念.....	61
10.1.1 概论 .....	61
10.1.2 相关概念 .....	61
10.2 线性表的检索.....	62
10.2.1 概论 .....	62
10.2.2 顺序检索 .....	62
10.2.3 二分检索 .....	63
10.2.4 分块检索 .....	63
10.3 树形结构的检索 .....	63
10.3.1 概论 .....	63
10.3.2 二叉排序树 .....	64

10.3.3 AVL 树 .....	64
10.4 散列表的检索 .....	66
10.4.1 概论 .....	66
10.4.2 散列函数 .....	67
10.4.3 碰撞的产生及处理 .....	67
10.5 基于属性的检索 .....	68
参考文献 .....	69
附录 .....	70
致谢 .....	71
论文（设计）成绩 .....	72

# 第一章 数据结构绪论

## 1.1 数据结构的基本概念及研究内容

数据元素：具有完整确定意义的描述现实的某一个客观实体的一个最小数据集。数据元素类似原子，可以再分，每一项被称作数据项。

数据对象：具有相同属性的数据元素的集合。

数据结构：给定数据对象及其上面定义的操作所共同构成的一个系统一个信息处理模型。

主要研究的三个方面：

### 1. 数据的逻辑结构

- 逻辑关系

在自然形态下，数据元素之间的一种关系

- 逻辑结构

数据之间所有关系的一个集合数学表示： $B=(K,R)$ ，其中：K：数据上的有穷集合 R：K 上关系的有穷集合，其中每个关系 r 都是从 K 到 K 的关系

- 分类

– 线性

    一对一，单对单

– 非线性

    \* 树形结构

        唯一一个直接前驱，多个直接后继

    \* 图结构

### 2. 数据的存储结构

- 存储关系

存储关系的数学内涵：须要建立数据对象 (K) 到存储区域 (M) 的映射关系 (S):

$S:K \rightarrow M$

即  $\forall k \in K$ , 都有唯一的  $\forall Z \in M$ , 使得  $S(k)=Z$ , Z 为 K 结点所占存储空间的始单元。

- 存储结构

- 顺序结构  
按照连续地址空间的顺序依次的存放数据
- 链接结构  
存储密度相比顺序结构下降
- 索引结构
- 散列结构  
根据节点的值，通过一定的函数关系来确定数据元素的存储地址

3. 数据的运算关系定义在逻辑结构上，在存储结构上实施，即：

- 抽象层面
  - 逻辑关系
  - 需求
- 实现层面
  - 存储关系
  - 运算关系
- 评价层面

三个层次五个要素

## 1.2 数据结构的选择与评价

评价标准：

- 时间需要量与时间效率
- 存储需要量与存储效率

## 第二章 算法

### 2.1 算法的定义

#### 2.1.1 概述

算法是一个问题的具体解决方案。

#### 2.1.2 定义

算法解决某一个问题的指令的有限集合。

基本特征：

- 有穷性
- 确定性
- 可行性
- 输入
- 输出

#### 2.1.3 内涵及分类

内涵体现在过程上。

- 一般过程
- 函数过程

强调结果

#### 2.1.4 算法与程序的异同

- 程序不是算法
- 程序不是算法程序不一定满足有穷性

### 2.2 算法的描述及设计原则

#### 2.2.1 算法描述方法

- 计算机程序设计语言  
优点与缺陷：设计出 = 实现出
- 自然语言缺陷：雍长和二义性

- PDL
- 流程图

### 2.2.2 算法基本标准

- 正确性
- 易读性
- 健壮性
- 高效性

## 2.3 算法分析概论及有效算法

### 2.3.1 概念

一般不需要知道精确的时间消耗，需要知道时间消耗的增长率大体在什么范围。

算法复杂性的阶：算法比较主要比较阶。

时间复杂性（时间渐进复杂性）：利用某算法处理一个问题规模为  $n$  的输入所需要的时间，记为  $T(n)$ 。

空间复杂性：利用某算法处理一个问题规模为  $n$  的输入所需要的存储空间，记为  $S(n)$ 。

阶：对一个正常数  $C$ ，一个算法在时间  $(n^2)$  内能处理规模为  $n$  的输入，则称此算法的时间复杂度是  $((n^2))$ ，读作“ $(n^2)$  阶”，即该算法的时间复杂度与  $(n^2)$  是同阶的。

如果对某一正常数  $C$ , 一个算法在时间  $Cn^2$  内能处理规模为  $n$  的输入, 则称此算法的时间复杂性是  $O(n^2)$ , 读作“ $n^2$  阶”, 即该算法的时间复杂性与  $n^2$  是同阶的。

若问题规模为  $n$  ( $n > n_0$ ,  $n_0$  为问题最小规模值), 即:

$$T(n) = Cn^2, \text{ 若 } f(n) = n^2, \text{ 有:}$$

$$\lim_{n>n_0 \text{ 且 } n \rightarrow +\infty} \frac{T(n)}{f(n)} = c$$

则:  $T(n)$ 、 $f(n)$  同阶

编写:蒙应杰

若一个算法时间复杂度为  $O((2^n))$ , 称其需要指数时间; 若是  $O((n^k))$ , 称其为多项式时间。当  $n$  非常大时两个时间差异非常大。

以多项式时间为界限的算法称为有效算法。如果一个问题不存在以多项式时间为界限的算法, 称为难解的(难解性问题)。

§ 2.3 算法分析概论及有效算法							Data Structures
	时间 复杂度	多项式时间算法与指数时间算法的运行时间对比					
		n=10	n=20	n=30	n=40	n=50	n=60
A算法	$n$	$10^{-7}$	$2 \times 10^{-7}$	$3 \times 10^{-7}$	$4 \times 10^{-7}$	$5 \times 10^{-7}$	$6 \times 10^{-7}$
	$n^2$	$10^{-6}$	$4 \times 10^{-6}$	$9 \times 10^{-6}$	$1.6 \times 10^{-5}$	$2.5 \times 10^{-5}$	$3.6 \times 10^{-5}$
	$n^3$	$10^{-5}$	$8 \times 10^{-5}$	$2.7 \times 10^{-4}$	$6.4 \times 10^{-4}$	$1.25 \times 10^{-3}$	$2.16 \times 10^{-3}$
	$n^5$	0.001	0.032	0.243	1.02	3.12	7.80
	$2^n$	$10^{-5}$	0.001	10.74	3.05小时	130.3天	366年
B算法	$3^n$	$5.9 \times 10^{-4}$	34.8	23.7天	3855年	$2 \times 10^6$ 世纪	$1.3 \times 10^{11}$ 世纪
	$n$ 为数据规模, 在每秒运行速度为1亿次的计算机上的运行时间(单位:秒)						

编写:蒙应杰

当  $n$  非常大时两个时间差异非常大。

§ 2.3 算法分析概论及有效算法							Data Structures
	提高计算机的速度对单位时间内可解的问题规模的影响						
	时间 复杂度	单位时间内可解的问题规模					
A算法	$n$	$N_1$	$100N_1$		$1000N_1$		
	$n^2$	$N_2$	$10N_2$		$31.6N_2$		
	$n^3$	$N_3$	$4.64N_3$		$10N_3$		
	$n^5$	$N_4$	$2.5N_4$		$3.98N_4$		
	$2^n$	$N_5$	$N_5 + 6.64$		$N_5 + 9.97$		
B算法	$3^n$	$N_6$	$N_6 + 4.19$		$N_6 + 6.29$		

从此表可看出, 把计算机速度提高1000倍, A算法在单位时间内可解问题的规模是原来的10倍, 而对于B算法, 问题规模只增加9.97( $\approx 10$ )个。

编写:蒙应杰

对于时间复杂度, 会关注时间复杂度的最坏情况和时间复杂度的平均情况。

### 2.3.2 复杂性分析

不需要知道精确的数值, 只需要知道他一个规模。

- 时间复杂性的分析

1. 根据问题的特点合理选择一种或几种操作作为整个算法的“标准操作”（假设循环执行次数最多，循环就是标准操作）
  2. 确定每个算法在给定输入下总共执行了多少次标准操作，并根据次数推导求出时间函数
  3. 确定该函数的阶
- 空间复杂性的分析
    1. 根据问题的特点合理选择一种或几种操作作为整个算法的“标准操作”（假设循环执行次数最多，循环就是标准操作）
    2. 确定每个算法在给定输入下总共执行了多少次标准操作，并根据次数推导求出时间函数
    3. 确定该函数的阶

要求各个算法的存储结构一样的前提下，关注算法所需的附加存储空间复杂性

## 2.4 算法设计方法概论

### 2.4.1 概述

- 用科学的方法进行算法设计
- 最常用方法：自顶向下逐步求精
  - 顶层：问题的总和、抽象全貌
  - 底层：问题的具体化、展开、细节
- 求精的方法：
  1. 分而治之
    - 将问题划分为一些不相交的部分，依次解决
  2. 作出有限进展
    - 采用一个朝解的方向得到有限进展的方法，反复应用，逐渐逼近
  3. 情况分析
    - 对问题各种情况给予分析，选择适当方案。
- 健壮性
- 高效性

### 2.4.2 算法设计的基本技术

以下内容不做要求

- 穷举法
- 分治法
- 回溯法
- 分支界限法
- 动态规划法
- 贪心法

## 2.5 算法描述语言

### 2.5.1 PDL 概述

Program、Design、Language

即伪码语言，主要用来书写软件设计的规约，是基于我们自然语言与具体的成设计语言之间的一种语言。这是一种保留计算机、程序设计、语言的基本框架和描述形式，并去掉一些特异性和直接性的要求，再结合自然语言所形成的一种用于描述算法处理的逻辑语言。

### 2.5.2 PDL 的优势

- 表达能力强，具有关键字的固定语法。提供了特定的结构化控制结构
- 引入了自然语言的一些习惯，结构比较清晰，简单易读
- 容易转化为任何一种程序设计语言代码（可由 PDL 生成程序代码）

### 2.5.3 PDL 书写及要求

#### 1. 算法的框架

- 一般过程的书写框架

```
1 PROC 过程名(I/O参数);  
2 BEGIN  
3     语句组  
4 END;
```

- 函数过程的书写框架

```
1 FUNC 函数名(I/O参数):类型名;  
2 BEGIN  
3     语句组  
4 END;
```

#### 2. 词的定义及说明

标识符：按照一定的规则形成的具有特定含义的一个词。

- 过程名：调用前需定义
- 常量名、变量名：使用前需说明

例如 VAR i,j,k:integer

常见的数据类型名写法及表示：

- 整数型：integer
- 实数型：real
- 布尔型：boolean
- 字符型（单字符）：char
- 子介型（用于表达范围）：下界..上界，例如 40..90(表示 40-90), 'A'..'G'
- 枚举类型：0,1,2,3 元素次序不能变
- 构造类型
  - \* 数组型：

```
1 ARRAY[ 下标类型 ] OF 成分类型
```

例如：

```
1 A: ARRAY[1..20] OF integer
```

```
1 B: ARRAY[1..20, -10..20] OF real
```

```
2 // (B是点集，二维数组)
```

\* 记录型：

```
1 RECORD
```

```
2     域标识符1：类型1
```

```
3         ...
```

```
4     域标识符n：类型n
```

```
5 END
```

例如：

```
1 A=RECORD
```

```
2     Name:ARRAY[1..8] OF char;
```

```
3     Sex:0..1
```

```
4     Age:integer;
```

```
5 END
```

– 指针类型:

1      类型名

例如:

1 TYPE A= integer;    //指针类型

1 VAR B: integer;    //指针变量

### 3. 基本语序

• 赋值语句:

1 变量名 表达式

• 流程图

• 条件语句

– 形式一

1 if 条件 then 语句组

– 形式二

1 if 条件 then 语句组1

2                        else 语句组2

• 循环语句

– 当型 (while)

1 WHILE 条件 DO

2                        语句组;

– 直到型 (repeat)

1 REPEAT

2                        语句组;

3 UNTIL 条件

– 从到型 (for-to)

\* 默认步长为 1:

1 FOR 变量 初值 TO 终值 DO

2                        语句组;

\* 自定义步长:

```

1 FOR 变量 初值 TO 终值 STEP 步长值 DO
2     语句组;

```

\* 倒数:

```

1 FOR 变量 初值 DOWNTO 终值 DO
2     语句组;

```

- 输入语句:

```
1 read(变量名表);
```

例:

```
1 read(x,y,z);
```

- 输出语句

```
1 write(变量名表);
```

#### 4. 拓展语序

- 情况语句

```

1 CASE
2     条件 1: 语句组 1;
3     条件 2: 语句组 2;
4     .....
5     条件 n: 语句组 n;
6     [ELSE 语句组 n+1]
7 ENDCASE

```

- 一般过程调用语句:

```
1 Call 过程名;
```

- 函数过程调用: 通过在表达式中引用函数名完成, 即被引用函数名出现在表达式中

- 出错提示语句:

```
1 error(错误信息);
```

- 终结语句

```
1 Exit \\ 算法转向正常结束
```

```
1 Return \\ 算法转向正常结束, 携带值离开
```

1 Abort \\中途废止（中止）

- 复合语句

```
1 [ 简单语句1;  
2     简单语句2;  
3     .....  
4     简单语句n; ]
```

或用 Begin End 代替括号

- 动态符号

- 储存单元的引用:

1 指针变量名

例:

1 x

- 动态空间分配:

1 New(P)

- 动态空间回收:

1 Dispose(P)

- 空地址的表示:

1 Nil

## 第三章 线性表

### 3.1 线性表及其运算

#### 3.1.1 线性表的定义

线性表的定义：一个线性表是  $n \geq 0$  个数据元素  $a_1, a_2, \dots, a_n$  的有限序列，序列中除第一及最后一个元素以外，每个元素有且只有一个直接前驱和直接后继。

简称表，可表示为： $A = (a_1, a_2, \dots, a_n)$

```
1 ai:datatype //表示 ai 项可以是任何类型
```

#### 3.1.2 线性表的特征

- 有限的。线性表的表长：线性表元素的个数。控标的长度定义为 0
- 元素呈线性关系。元素的位置只取决于他们自己的逻辑顺序。

#### 3.1.3 线性表的运算

- 确定线性表的长度  $n$
- 存取线性表的第  $i$  个数据元素，检验或改变某个数据项的值
- 在第  $i-1$  个和第  $i$  个数据元素之间插入一个新的数据元素。约定插入的元素是第  $i$  个元素的直接前驱
- 删去第  $i$  个元素
- 将两个或两个以上的线性表合并成一个线性表
- 将一个线性表拆分成两个或两个以上的线性表
- 重新复制一个线性表
- 对线性表中的数据元素依据某一种规则进行重组

### 3.2 线性表的储存表示

- 线性表的向量表示

- 存储方法顺序地分配存储单元，且每个数据元素占据相同大小的存储空间（顺序且等长）
  - 数据访问 TODO

- 向量存储结构特性
  - \* 储存分配呈线性结构
  - \* 属于随机存储结构（访问一个元素的代价与元素位置无关，即访问任何一个元素（找地址）的运算量一样，一维数组，通过下标变量来访问（数组构造的本质即算公式））

- 向量存储结构的形式化表示可用一个一维数组表示，由于数组属于静态结构，其空间规模须事先定义（1..max），要有个计数器记录数组长度（空间规模）

表述形式：

```

1 TYPE SQLIST:ARRAY [ 1 .. max ] OF datatype ;
2 ## 内容
3 VAR n:0 .. max;
4 ## 计数器（记录数组长度）

```

```

1 TYPE SQLIST= RECORD
2             data:ARRAY [ 1 .. max ] OF datatype ;
3 ## 内容 n:0 .. max; ## 计数器（记录数组长度）
4             END;

```

- 插入
- 删除
- 小结

- 线性表的链表表示

- 单链表表示

```

1 TYPE pointer = node ;
2     node= RECORD
3             data: datatype ;
4             next: pointer ;
5         END;
6         link=pointer ;

```

- 带表头的单链表表示
- 带表头结点的循环单链表表示

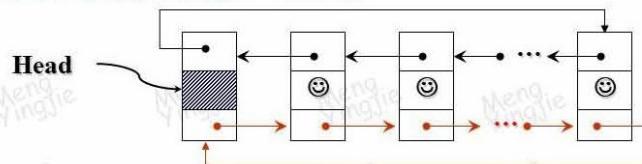
- 带表头结点的双向循环链表表示

```

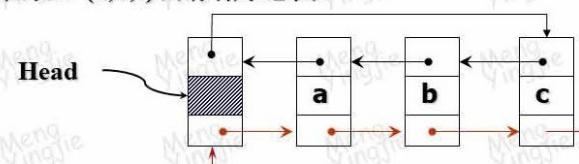
1 TYPE pointer = node;
2     node= RECORD
3             Left: pointer;
4                 data: datatype;
5
6             right: pointer;
7         END;
8     dblink=pointer;

```

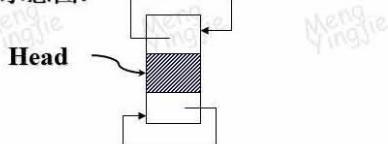
**带表头的双向循环链表的一般表示:**



例1，线性表A=(a,b,c)的存储示意图:

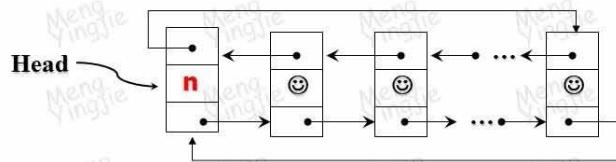


例2，线性表A=() 的存储示意图:

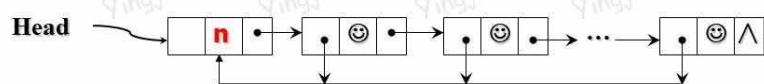


实际使用中双向链表的一些其它变化:

**例1. 表头结点的数据项的利用:**



**例2. 指针数据项的变化:**



- 线性表的应用

## 第四章 栈和队列

### 4.1 栈及其运算

栈是工具性数据结构。

#### 4.1.1 绪论

#### 4.1.2 栈的基本定义

栈是一个下限为常数，上限可变化的向量（或者反之）。有时称为堆栈或堆阵。

后进先出表（LIFO）。

可变化一端称为栈顶，不变化的一端称为栈底。

#### 4.1.3 与线性表的关系

- 相同点

- 逻辑关系都是线性关系

- 不同点

- 线性表可以从任意位置增删，栈的增删只能从表尾进行
  - 栈的运算是线性表的一个子集，且这个子集还需加以约束

#### 4.1.4 栈的运算

- 入栈，PUSH(S)，完成往栈中加入元素的过程，即插入操作，也称为压栈
- 出栈，POP(S)，完成从栈中取出元素的过程，即删除操作，也称为弹出

#### 4.1.5 栈的存储与运算的实现

#### 4.1.6 多栈共存问题

### 4.2 栈的应用

### 4.3 队列及其运算

- 绪论分时和并行，主要处理对稀缺资源的争夺
- 队列的定义队列是一个下限和上限只能增加而不能减少（下限和上限的指针只能往一个方向移动）的向量（或者反之）

### 先进先出表 (FIFO)

- 队列与线性表
  - 相同点
    - \* 逻辑关系都是线性关系
  - 不同点
    - \* 线性表可以从任意位置增删，队列的只能一端插入一端删除
    - \* 队列的运算是线性表的一个子集，且这个子集还需加以约束
- 队列的运算
  - 出队
  - 入队
- 队列的存储与运算的实现

## 4.4 受限的栈及队列（了解）

- 双端队列

双端队列是一种所有的插入和删除都限制在表的两端进行的线性表

- 双栈

双栈是一种加限制的双端队列，即从哪端进就只能从哪端出，就像是两个底部相连的栈

- 超队列

超队列是一种删除受限制的双端队列，删除限制在一端，插入可以在两端

- 超栈

超栈是一种插入受限制的双端队列，插入限制在一端，删除可以在两端

## 第五章 串

### 5.1 串及其运算

- 概述

字符串是线性表模型数据元素实例化的体现

全称字符串

早期：作为输入输出的常量和提示

热点：中文信息处理

- 串的定义

一个由零个或多个字符组成的有穷序列称为串

简记为  $A = 'a_1a_2a_3 \dots a_n'$

串的长度：串中所含的字符个数

空串：串长为零的串，记作

非空串

空白串：空白字符组成的串，记作 ''

串的相等：串长也相等，对应位置上的字符也一样

字串/主串：一个串中任意个连续字符组成的子序列称为该串的子串，该串成为它的所有子串的主串。不一定是一个真子集

- 串的运算

- 赋值

assign(S,chars) 将字符串常量 chars 赋给字符串变量 S

- 连接

concatenation(S,T) 将字符串 S 和 T 联接在一起

- 取子串

substring(S,m,n) 从第 m 个位置开始取 n 个连续字符（通常）/从第 m 个位置开始到第 n 个位置

- 求子串序号

strindex(S,T,i) 确定串 T 在 S 中第一次出现的位置 i

- 串的插入

strinsert(S,T,i)

- 串的删除
  - strdelete(S,m,n)
- 串的复制
  - copy(S,T)
- 串的置换
  - replace(A,B,C) (找到 A 中的 B 用 C 来替换)
  - replace(S,m,n,T) (把 S 字符串的 (m 到 n/从 m 开始的 n 个字符) 用 T 来替换)
- 串的存储
  - 顺序储存
    - 一般用向量来表示，通常称为字符串数组、字符串变量、字符串常量
    - \* 压缩模式
      - 按字节存储数据 (C 语言)
    - \* 非压缩格式
      - 运算器以字为单位运算，存储器以字节为单位，为了避免转换浪费的时间，存储器以字为单位存储一个字符，变成非压缩模式 (只存一个字母需要一个字的空间，比较浪费)
  - 索引储存
    - 多用在对于多个字符串常量或者变量的组织
  - 链接储存
    - 理论上研究的多，实际上用的少，不易于实现

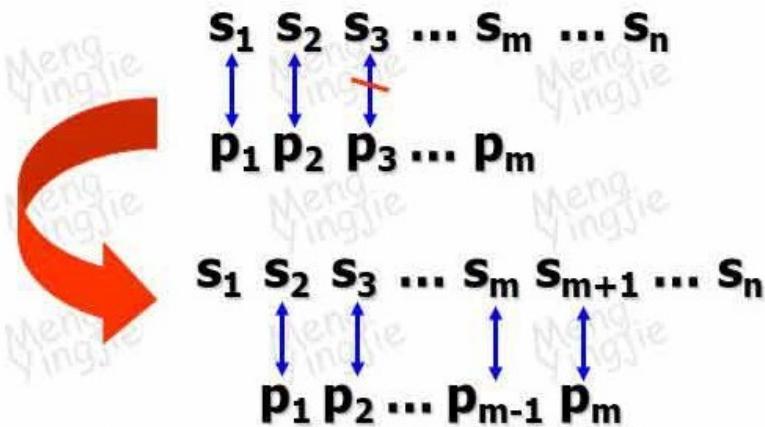
## 5.2 串的模式匹配

- 概述

在模式分类或者问题回答系统等方面，将输入模式与样本模式进行匹配的过程啊，我们就把它称为模式匹配。

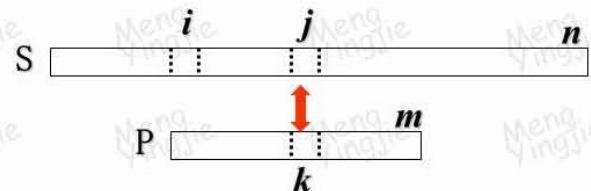
通常把 S 称为目标 (或正文)，把 P 称为模式，把从目标 S 中查找模式 P 的过程称为模式匹配。

串的匹配是模式匹配的实例化。
- 匹配的朴素算法 (Brute-Force 算法)



处理基本思想：对正文顺序搜索

1. 逐次比较（对应字符依次比较）
2. 发现不匹配时，将 P 相对于 S 右移一位
3. 重复上述过程，直到成功或扫描完 S



```

PROC Search(VAR S,P:string;i:intger);
BEGIN i←0;
  WHILE i<n DO
    [ i←i+1 ; j←i ; k←1;
      WHILE S[j]=P[k] DO
        IF k=m THEN [ WRITE(i,'success'); exit ]
        ELSE [ j←j+1 ; k←k+1 ] ;
      WRITE('failure')
    END;
  
```

优化算法：KMP 算法

### 5.3 例题

2019 年兰州大学开源社区纳新面试题中有一题关于字符串的运算，现将我的解决方案附于后文。

给出一个文本文件，其中每个单词不包括空格及跨行，单词由字符序列构成且不区分大小写，完成以下功能：统计给定单词在文本文件中出现的总次数。

注：允许使用 string.h

```
1 #include<stdio.h>
2 #include<string.h>
3
4 char word[100];
5 char text[10000];
6 int count = 0;
7
8 char trans(char c){
9     if(c >= 'A' && c <= 'Z'){
10         c = c + 'a' - 'A';
11     }
12     return c;
13 }
14
15 void getword() {
16     char c;
17     int i = 1;
18     while (1) {
19         c = getchar();
20         if (c == '\n') break;
21         word[i] = trans(c);
22         i++;
23     }
24     word[0] = i - 1; //第0位存放单词长度
25     return;
26 }
27
28 void gettext() {
29     char c;
30     int i = 1;
31     while (1) {
32         c = getchar();
33         if (c == EOF) break;
34         text[i] = trans(c);
```

```
35         i++;
36     }
37     text[0] = i - 1; //第 0 位存放文本长度
38     return;
39 }
40
41 void check() {
42
43     int i = 0, j = 0;
44     int flag = 1;
45     while (1) {
46         i++;
47         j++;
48         if (i == text[0]) break;
49         if (text[i] == '\n' || text[i] == ' ') {
50             j = 0;
51             flag = 1;
52             continue;
53         }
54         if (flag == 0) {
55             continue;
56         }
57         if (text[i] != word[j])
58             flag = 0;
59         if (j == word[0]) {
60             if (flag == 1) {
61                 if (text[i + 1] == '\n' || text[i + 1] == ' ')
62                     count++;
63             }
64             flag = 1;
65         }
66     }
67 }
```

```
68
69 int main()
70 {
71     freopen( "C:\\\\Users\\\\Kente\\\\Desktop\\\\123\\\\text.txt" , "r"
72         , stdin); //假设给出的文本文件第一排是给定单词，第二
73         //排及之后是要统计的文本
74     getword();
75     gettext();
76     check();
77     printf( "%d" , count);
78     return 0;
79 }
80 /*text.txt
81 ABC
82 Abc abC dasadsa abc
83 dasdasdasdasd dasdasdasdasd
84 abc ab
85 */
```

## 第六章 数组和广义表

### 6.1 数组的定义与运算

#### 6.1.1 定义

一维数组是一个向量，它的每个元素是该结构中不可分割的最小单位； $n(n>1)$  维数组是个向量，它的每个元素是  $n-1$  维数组，且具有相同的下限和上限。

向量（只有量值而无方向的量）是标量的一维的有序集合。

#### 6.1.2 运算

- 给定一组下标，存取相应数据元素
- 给定一组下标，修改相应的数据元素的某个属性的值

### 6.2 数组元素的地址访问

#### 6.2.1 概述

数组是一种静态存储结构，数组定义需给出数组规模。数组元素访问依靠下标变量。

词典编辑序：

- 行主序（行优先，行为主）（更常见）  
从上到下，每行从左到右
- 列主序  
从左到右，每列从上到下

$$\begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & \mathbf{a}_{13} & \mathbf{a}_{14} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} & \mathbf{a}_{24} \\ \mathbf{a}_{31} & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{a}_{34} \end{pmatrix}$$

存储空间中的存放顺序：

行主序：

$$\begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & \mathbf{a}_{13} & \mathbf{a}_{14} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} & \mathbf{a}_{24} \\ \mathbf{a}_{31} & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{a}_{34} \end{pmatrix}$$

列主序：

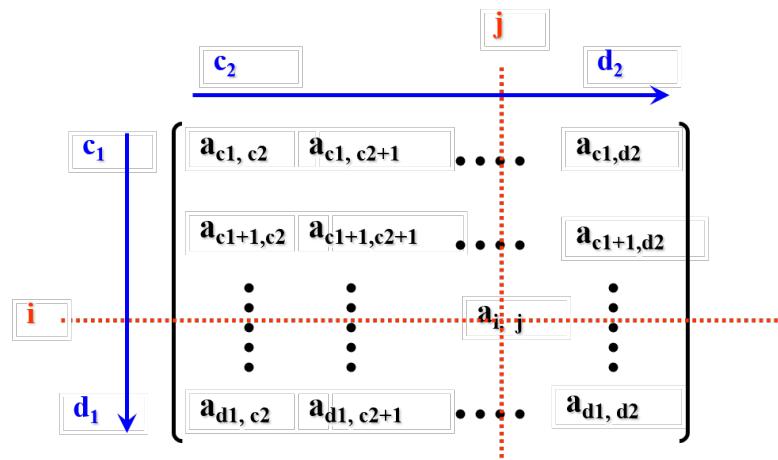
$$\begin{pmatrix} \mathbf{a}_{11} \\ \mathbf{a}_{21} \\ \mathbf{a}_{31} \\ \mathbf{a}_{12} \\ \mathbf{a}_{22} \\ \mathbf{a}_{32} \\ \mathbf{a}_{13} \\ \mathbf{a}_{23} \\ \mathbf{a}_{33} \\ \mathbf{a}_{14} \\ \mathbf{a}_{24} \\ \mathbf{a}_{34} \end{pmatrix}$$

## 6.2.2 地址计算

(默认讨论行主序)

### 二维数组

设置数组  $A(c_1, d_1, c_2 \dots d_2)$ , 首地址为  $\text{Loc}(c_1, c_2) = AO(A)$  (基地址), 元素长度为 1



$$\text{Loc}(i, j) = [(i - c_1)(d_2 - c_2 + 1) + (j - c_2)] * l + AO(A)$$

二维数组是随机存储结构。

### 多维数组

设置数组  $A(c_1 \dots d_1, c_2 \dots d_2, \dots, c_n \dots d_n)$ , 首地址为  $\text{Loc}(c_1, c_2, c_3, \dots, c_n) = AO(A)$  (基地址), 元素长度为 1。

$$\begin{aligned}
 \text{Loc}(j_1, j_2, \dots, j_n) &= \text{AO}(A) + l [ (j_1 - c_1) \times (d_2 - c_2 + 1) \times (d_3 - c_3 + 1) \times \dots \times (d_n - c_n + 1) + \\
 &\quad (j_2 - c_2) \times (d_3 - c_3 + 1) \times \dots \times (d_n - c_n + 1) + \dots + \\
 &\quad (j_{n-1} - c_{n-1}) \times (d_n - c_n + 1) + (j_n - c_n) ]
 \end{aligned}$$

$$= \text{AO}(A) + l \left[ \sum_{i=1}^{n-1} (j_i - c_i) \prod_{k=i+1}^n (d_k - c_k + 1) + (j_n - c_n) \right]$$

$$\begin{aligned}
 \text{Loc}(j_1, j_2, \dots, j_n) &= \text{AO}(A) + l \left[ \sum_{i=1}^{n-1} (j_i - c_i) \prod_{k=i+1}^n (d_k - c_k + 1) + (j_n - c_n) \right] \\
 &\quad \downarrow \text{其中 } \begin{cases} \alpha_i = l \prod_{k=i+1}^n (d_k - c_k + 1), 1 \leq i \leq n-1 \\ \alpha_i = l, i = n \end{cases} \\
 &= \text{AO}(A) + \sum_{i=1}^n \alpha_i (j_i - c_i) \\
 &= \text{AO}(A) - \sum_{i=1}^n \alpha_i c_i + \sum_{i=1}^n \alpha_i j_i \\
 &\quad \text{VO}(A) = \text{AO}(A) - \sum_{i=1}^n \alpha_i c_i \\
 &\quad \text{虚拟地址} \\
 &\quad \downarrow \\
 \text{Loc}(j_1, j_2, \dots, j_n) &= \text{VO}(A) + \sum_{i=1}^n \alpha_i j_i
 \end{aligned}$$

多维数组是随机存储结构。

数组编译就是利用公式把虚拟地址运算出来。

地址访问公式中的常数项之和就是虚拟地址。

### 6.2.3 特殊数组

#### 三角矩阵

(行优先)地址访问:

前*i-1*行元素总数:

$$1+2+3+\dots+(i-1)$$

$$= i \times (i-1)/2$$

第*i*行元素个数:

$$(j-1)-1+1=j-1$$

$$\begin{aligned}
 \text{Loc}(a_{ij}) &= \text{Loc}(a_{11}) + [i \times (i-1)/2 + (j-1)] \times l \\
 &\quad 1 \leq j \leq i \leq n
 \end{aligned}$$

$$\begin{array}{cccccc}
 & & & j & & \\
 & & & \downarrow & & \\
 & & & a_{11} & 0 & \cdots & \cdots & 0 \\
 & & & a_{21} & a_{22} & 0 & \cdots & 0 \\
 & & & \cdots & \cdots & \cdots & \cdots & \cdots \\
 & & & a_{i1} & \cdots & a_{ij} & \cdots & \cdots \\
 & & & \cdots & \cdots & \cdots & \cdots & \cdots \\
 & & & a_{n1} & a_{n2} & \cdots & \cdots & a_{nn}
 \end{array}$$

### 三对角矩阵

$$\begin{bmatrix} a_{1,1} & a_{1,2} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} & \vdots \\ \vdots & & & & \vdots \\ \vdots & & & & a_{n-1,n} \\ 0 & \cdots & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

## 6.3 稀疏矩阵

### 6.3.1 概要

研究重点：研究特殊存储方法。

### 6.3.2 定义

#### 稀疏数组

在一个数组当中和某元素比较而言，不相同的元素很少时，我们称此数组为稀疏数组。

“某元素”指的是大量的雷同元素，可以认为是 0。

“很少”指的是远远小于。

#### 稀疏矩阵

在一个矩阵当中和某元素比较而言，不相同的元素很少时，我们称此矩阵为稀疏矩阵。

#### 特殊矩阵与稀疏矩阵的异同

特殊矩阵的雷同元素分布有规律，稀疏矩阵的雷同元素分布无规律。

### 6.3.3 稀疏矩阵的存储

按照行（或列）优先的原则，将矩阵中的非零元素顺序存放，为便于检索和存取，一般须带有适当的辅助信息。

#### 顺序存储

- 三元组（triad）表示
  - 两种实现方式
    - \* 视为一维的记录数组

	row	col	data
1	1	1	7
2	3	2	-4
3	3	5	5
4	4	4	6
5	5	1	1
6	5	5	3

- \* 视为二维数组

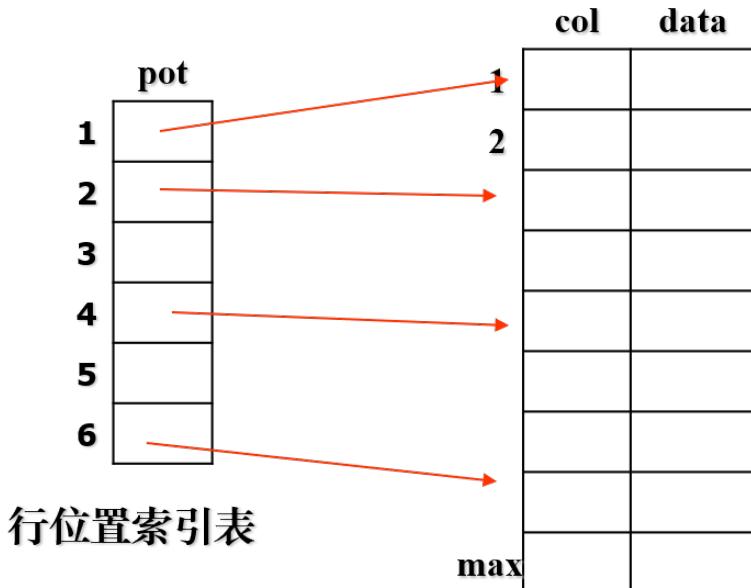
**A(0..p, 1..3),**  
其中 p 为非零元  
个数，第 0 行存  
放矩阵信息

	1	2	3
0	5	5	6
1	1	1	7
2	3	2	-4
3	3	5	5
4	4	4	6
5	5	1	1
6	5	5	3

row    col    data

- 缺陷
  - \* 用向量实现，属于静态结构
  - \* 元素规模事先确定
  - \* 插入删除运算才会带来大规模的元素的移动
- 索引表示（二元组表示）
 

三元组中数据元素按 row 呈有序状态（行优先次序决定），故可建立行索引表，即标记每行的非零元起始位置



- 伪地址表示
 

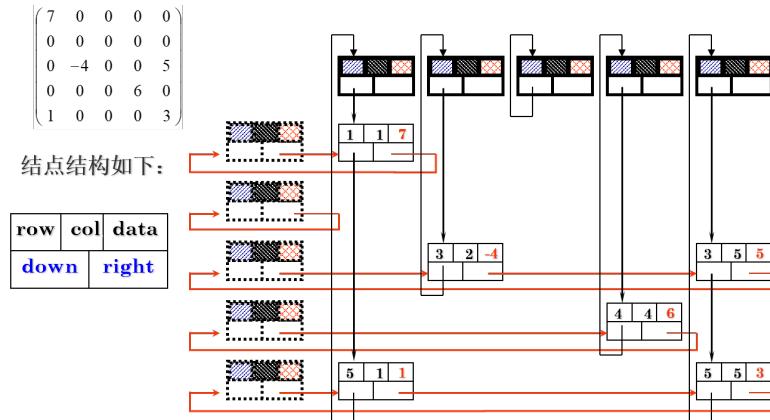
按照非零元在矩阵中出现的相对次序作为元素的地址映射关系来组织元素次序（在行或列优先的次序下）。

缺陷：计算之前先要用一下原有数组那个编译器当中那个计算公式计算它是第几个

## 链接存储

- 单链表
- 十字链表

对于非零元素依据行、列特性，每一行和每一列建立一个循环链表。



行表头（虚线）向下域没用处；列表头（实线）向右域没用处；行列表头节点可以共享。表头结点的数据项是没有用的。用表头结点的数据项把所有的表头结点串起来。此时数据项称为共享（C 中的共用体）。

## 散列存储

### 6.3.4 稀疏矩阵的运算

保持结构的一致性。

## 6.4 广义表（了解）

### 6.4.1 定义

**自然语言：**

广义表是零个或多个原子或子表所组成的有限序列。一般简称为列表或表。

**形式化表示为：**  $A = (a_1, a_2, \dots, a_i, \dots, a_n)$

组成一个表的元素  $a_i$  可以是原子（即数据元素），也可以是子表。

原子是数据元素，是我们要处理的真实的客观事物，是一个确定的概念。

子表是作为构成元素的表，这是一个递归定义。

**广义表的表长：**

表包含的元素的个数。要素可以是原子数据，也可以是子表。

长度为 0 的表为空表。

**广义表的深度：**

出现的表中套表这种嵌套的层数目的最大值。

大写字母表示集合、一堆数据；小写字母表示具体客观事物。

$$\mathbf{F} = (\mathbf{a}, \mathbf{b}, (\mathbf{c}, (\mathbf{d})))$$

**a,b 在第一层**

**c 在第二层**

**d 在第三层。**

**F的长度为 3， 深度为 3 .**

举例：

$$E = ()$$

$$L = (a, b), \text{ 或 } L(a, b)$$

$$A = (x, L) = (x, (a, b)), \text{ 或 } A(x, L(a, b))$$

$$B = (A, y) = ((x, (a, b)), y)$$

$$C = (A, B) = ((x, (a, b)), ((x, (a, b)), y))$$

$$D = (z, D) = (z, (z, (\dots))))$$

$$F = ((r, s, t))$$

#### 6.4.2 特性

定义是递归的

定义没有限制元素的共享性和递归性

- 共享性：组成要素可以是任何现存的已经存在的表
- 递归性：广义表可用自身作为元素使用（可能导致问题）

#### 6.4.3 存储

顺序存储方式

链接存储方式

- 等长模式
- 非等长模式

## 第七章 树形结构

### 7.1 树的基本定义和运算

#### 7.1.1 树形结构的概述

- 与线性结构的差异  
直接后继可以是多个
- 树形结构的地位  
树是计算机中最重要的非线性结构
- 讨论内容
  - 逻辑角度
  - 存储方式
  - 应用
  - 模型
    - \* 树
    - \* 二叉树

#### 7.1.2 树的基本定义

树  $T$ , 是满足如下性质的有限个节点组成的非空集合

- $T$  中有且仅有一个称为根的结点
- 除根结点之外, 其余结点分成  $m(m>0)$  个不相交的集合  $T_1, T_2, \dots, T_m$ , 其中每个  $T_i$  都是树, 而且都称为  $T$  的子树。

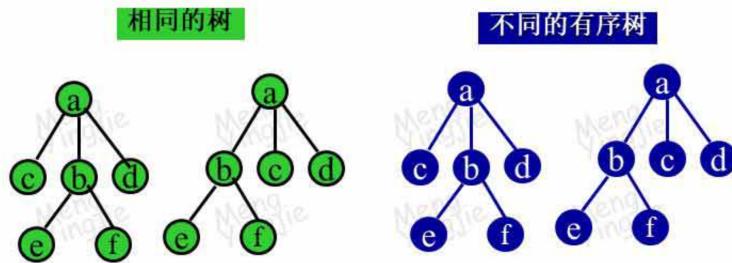
注意

- 是递归定义
- 定义中对于子树的个数及次序没有进行任何约束
- 与图论中的树不同, 是有向图中的根树的特例。隐含了向下的方向性

#### 7.1.3 有序树的基本定义

在树  $T$  中如果子树  $T_1, T_2, \dots, T_n$  的相对次序是重要的 (即有序的), 则称  $T$  为有向有序树, 简称有序树。

默认树是有序树



#### 7.1.4 森林（树林）

森林是零棵或多棵不相交的树的集合（通常是有序集合）

## 7.2 二叉树

- 定义
- 特性
- 二叉树的存储表示

## 7.3 遍历二叉树

### 7.3.1 概念

定义：对于给定的数据结构，系统地访问该结构中的每个结点，且每个结点仅被访问一次的操作过程称遍历

系统地：按照一定的规律、次序

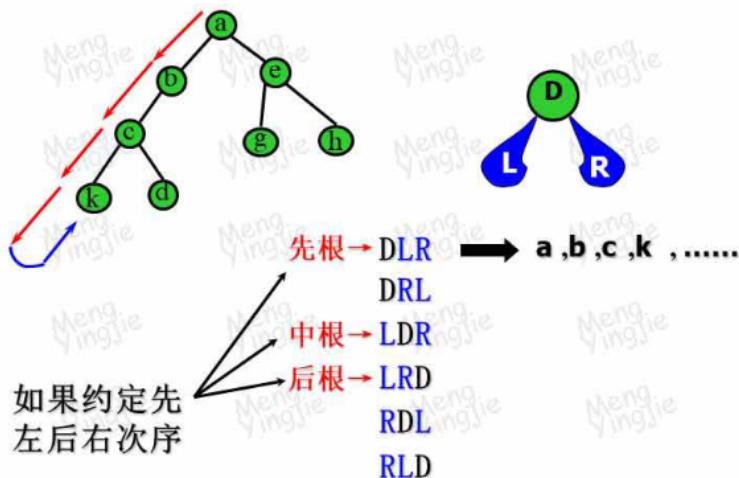
访问：对元素进行的某种操作

遍历本质上是一个运算过程（序列）

对于给定的二叉树，系统地访问该结构中的每个结点，且每个结点仅被访问一次的操作过程称遍历

### 7.3.2 二叉树的遍历次序（order）

- 层次策略
  - 自上而下
    1. 从左到右
    2. 从右到左
  - 自下而上
- 深度策略遍历针对结构，是一个过程；访问针对一个点，是一个动作。



### - 先根（前序）DLR

先拿到根，然后从左子树走，又是二叉树，则访问了左子树的根节点，再往下走，直到访问到 k，左子树为空，回到 c，访问右子树。沿着纵深往下搜索，只要有子树就往下搜索，先不管兄弟。

### - 中根 LDR

### - 后根（后序）LRD

## 7.4 树、森林与二叉树的转换

### 7.4.1 概述

转换原因

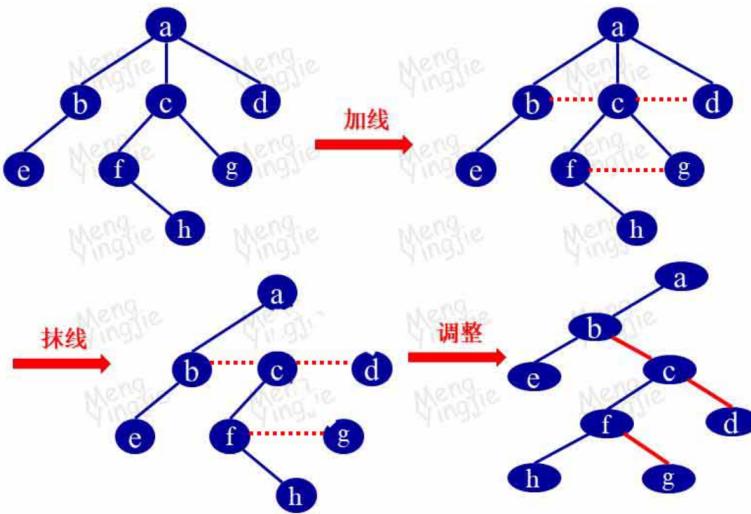
- 二叉树有许多优良特性，而树没有
- n 叉树中二叉树的空间利用率最高

### 7.4.2 树转为二叉树

这里约定树为有序树，子树的次序相对固定

转换规则：

1. 加线（亲兄弟之间加上虚连线）
2. 抹线（除了最左子节点之外，抹掉所有节点与其余子节点的连线）
3. 调整（调成二叉树的样子，原有连线在左，新加连线在右）



特点：

- 原有兄弟成为右结点及右结点的右列结点
- 根子树仅有左节点

#### 7.4.3 二叉树还原为树

转换规则：

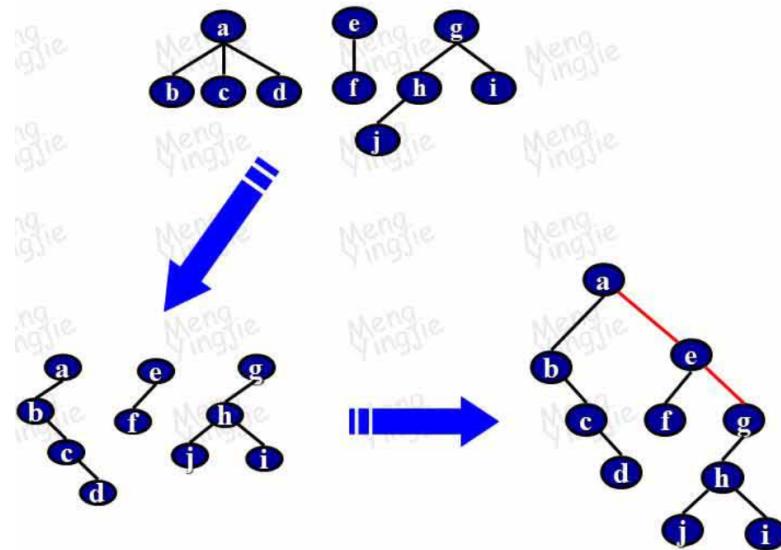
1. 加线（如果有节点 X 是父结点的左子树，那么它的右子树，右子树的右子树... 都与 X 的父节点加线）
2. 抹线（所有节点抹掉与右子树的连线）
3. 调整（让节点按层次分布）

#### 7.4.4 森林转为二叉树

两种方法：

- 方法一：树转换  
将每棵树转为二叉树
- 方法二：二叉树连接

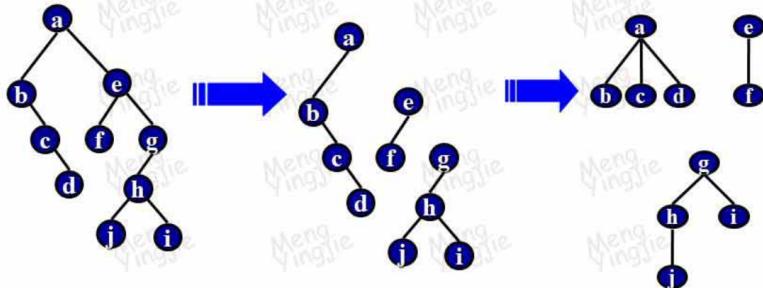
依据转换得到的二叉树的次序，将后一棵作为前一棵根节点的右子树，因为树转二叉树的根节点无右子树



#### 7.4.5 二叉树转为森林

转换规则：

1. 抹线（沿着根节点，抹掉其右子树与根节点的连线，不断往右下搜索）
2. 还原（把每棵二叉树还原树）



#### 7.4.6 树的遍历

- 方法一：树转换

将每棵树转为二叉树后再处理，处理完成后还原

- 方法二：直接处理

遍历次序

– 层次策略

\* 自上而下

1. 从左到右
2. 从右到左

\* 自下而上

– 深度策略

\* 先根（前序）DLR

1. 访问树的根节点
2. 先根次序下依次遍历树的根节点的每个子树

\* 中根 LDR

一般不讨论

\* 后根（后序）LRD

1. 后根次序下依次遍历树的根节点的每个子树
2. 访问树的根节点

#### 7.4.7 森林的遍历

- 方法一：树转换

将每棵树转为二叉树后再处理，处理完成后还原

- 方法二：直接处理

遍历次序

– 层次策略

\* 自上而下

1. 从左到右
2. 从右到左

\* 自下而上

– 深度策略

\* 先根（前序）DLR

1. 访问第一棵树的根节点
2. 先根次序下依次遍历第一棵树的根节点的每个子树
3. 先根次序下依次遍历其余的树

\* 中根 LDR

一般不讨论

\* 后根（后序）LRD

1. 后根次序下依次遍历第一棵树的根节点的每个子树
2. 访问第一棵树的根节点
3. 后根次序下依次遍历其余的树

## 7.5 线索树

## 7.6 树形结构的应用

## 第八章 图结构

### 8.1 基本概念

#### 8.1.1 概述

与线性结构和树形结构的差异

- 一个元素可以有多个直接前驱和后继
- 是多对多的数据关系

地位

图结构非常重要的非线性结构。

讨论内容

- 计算机中的图的表示方法
- 图的遍历
- 求生成树
- 找最短路径
- 拓扑排序
- 求关键路径

#### 8.1.2 概念

图

由  $n (n \geq 1)$  个结点  $\{v_1, v_2, \dots, v_n\}$  构成的数据  $G$  称为图，若结点集  $V = \{v_1, v_2, \dots, v_n\}$  上定义的称为后继的关系  $E$  是非自反的。

可表示为  $G=(V,E)$ ， $V$  称为顶点集， $E$  称为边集。只相当于

研究的图相当于离散数学中的简单图，相当于简单图，即不包含重边和环（自回路），不研究：

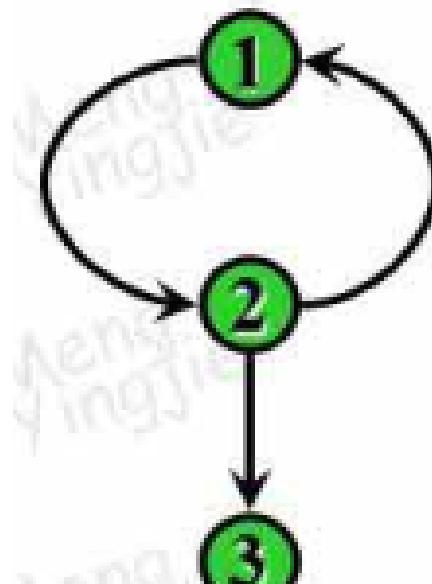
- 带自身环的图
- 多重图

有向图

在图  $G$  中，若每个关系都是顶点的有序对，则称  $G$  为有向图。

有向图的表示有两种方法：

- 形式化（符号）表示  
用有向线段指明了次序（方向）。
- 图形描述  
一般用尖括弧表示顶点的有序对。



图G:

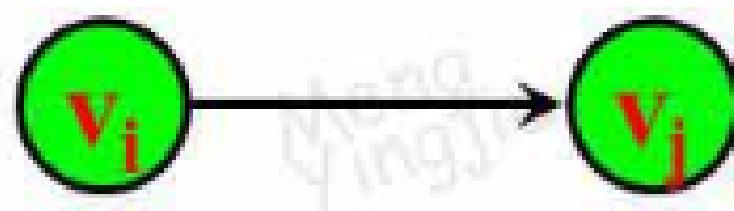
$$\begin{aligned} V(G) &= \{v_1, v_2, v_3\}, \\ E(G) &= \{\langle v_1, v_2 \rangle, \langle v_2, v_1 \rangle, \langle v_2, v_3 \rangle\} \end{aligned}$$

(a) 形式化（符号）表示

(b) 图形描述

图 8.1 图的表示方法

在有向图中，若  $\langle v_i, v_j \rangle \in E(G)$



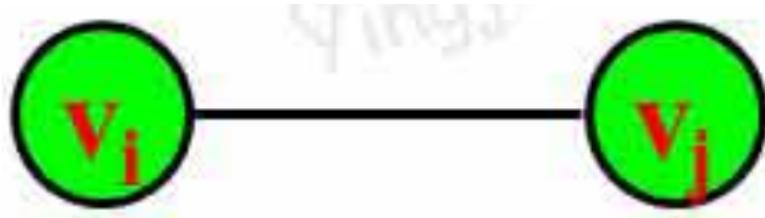
则称  $v_i$  是边的始点（或尾）； $v_j$  是边的终点（或头）；并称  $v_i$  邻接到  $v_j$ ， $v_j$  是从  $v_i$  邻接过来的；称边  $\langle v_i, v_j \rangle$  关联（或依附）于顶点  $v_i$  和  $v_j$ 。

推论：在  $n$  个结点的有向图中，其最大边数为  $n(n-1)$ 。把等于  $n(n-1)$  条边的图称有向完全图。

## 无向图

在图 G 中, 如果每个关系都是顶点的无序对, 则称 G 为无向图。有相图的表示有两种方法:

- 形式化(符号) 表示  
用无向线段表示(方向)。
- 图形描述  
一般用圆括弧表示顶点的无序对  
在无向图中, 若  $(v_i, v_j) \in E(G)$



则称  $v_i$  和  $v_j$  是邻接的; 并称边  $(v_i, v_j)$  关联(或依附)于顶点  $v_i$  和  $v_j$ .

在无向图中  $(v_1, v_2)$  和  $(v_2, v_1)$  这两个结点偶对表示同一条边。

推论: 在 n 个结点的无向图中, 其最大边数为  $n(n-1)/2$ 。把等于  $n(n-1)/2$  条边的图称无向完全图。

## 顶点的度、入度、出度

在图中, 顶点的度(degree), 就是与该顶点关联的边的数目。

若 G 为无向图, 则度为无向图中与某元素有关的元素的数量。

若 G 为有向图, 则

- 把以  $v_i$  为终点(头)的边的数目称作  $v_i$  的入度(incoming degree,in-degree)(即指向该元素的边数)
- 把以  $v_i$  为始点(尾)的边的数目称作  $v_i$  的出度(out going degree)(即从该元素指出去的边数)

在有向图中, 出度为 0 的顶点称终端顶点(叶子)。

推论: 设图 G 有 n 个结点, t 条边, 若  $d_i$  为顶点  $v_i$  的度, 显然有: TODO

## 子图

设图  $G = (V, E)$ , 如果有图  $G' = (V', E')$ , 且  $E'$  包含于  $E$ ,  $V'$  包含于  $V$ , 则称  $G'$  为 G 的子图。

如果图 G 的子图包含 G 的所有顶点, 则该子图称为 G 的生成子图。

自己可以是自己的子图 v。

## 路径、回路、图根

对于图  $G=(V,E)$ , 若存在结点序列  $v_p, v_{i1}, v_{i2}, \dots, v_{im}, v_q$  使得  $(v_p, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{im}, v_q)$  都在  $E(G)$  中 (对于有向图使得  $\langle v_p, v_{i1} \rangle, \langle v_{i1}, v_{i2} \rangle, \dots, \langle v_{im}, v_q \rangle$  都在  $E(G)$  中), 则称从顶点  $v_p$  到  $v_q$  存在一条路径 (或通路, path), 路径长度定义为这条路径上边的数目。(路径: 一个特定的点边交替序列)。

$v_p$  到  $v_q$  的路径可以简写为:  $v_p, v_{i1}, v_{i2}, \dots, v_{im}, v_q$  (用结点序简写)。

如果一条路径上的顶点除  $v_p$  和  $v_q$  可以相同以外其它顶点都不相同, 则此路径为一简单路径。(即中间没有重复经过的点)。

把  $v_q=v_q$  的简单路径, 称作回路 (或环, loop)。(不包含自身环)。

若在一个有向图中存在一个顶点  $v_0$ , 从此顶点有路径可以到达图中其它所有顶点, 则此有向图称为是有根的图,  $v_0$  称作图根。(树是有向图中的有根图的特殊情况)

## 连通性

- 无向图

两个顶点的连通: 在无向图  $G$  中, 顶点  $u$  和  $v(uv)$  之间存在一条路径, 则称顶点  $u$  和顶点  $v$  是连通的。

图的连通: 对于无向图  $G$  中的任意两个顶点  $u, v(uv)$  都是连通的则称此无向图是连通的, 或称  $G$  为连通图。(仅有无向图才谈图的连通)。

最大 (或极大) 连通子图: 无向图的连通分量。连通分量是一个 (或多个) 子图, 不是一个数字。

- 有向图

两个顶点的连通: 在有向图  $G$  中, 顶点  $u$  和  $v(uv)$  之间存在一条从  $u$  到  $v$  和  $v$  到  $u$  的 (有向) 路径, 则称顶点  $u$  和顶点  $v$  是连通的。(就是两个点能互相到达)

图的强连通: 有向图  $G$  中的任意两个顶点  $u, v(uv)$  都是连通的则称此有向图是强连通的, 或称  $G$  为强连通图。

一个有向图的强连通分量定义为该图的最大 (或极大) 强连通子图 (的个数)。(有向图不谈图的连通, 只谈图的强/弱连通)

## 网

给图的每一条边加一个 (非负的) 数, 这个与图的边相关的数值称为权 (weight)。带权的图称为网。带权的连通图称为网络。(网络针对无向图)

## 8.2 图的存储

### 8.2.1 邻接矩阵

定义：表示顶点之间邻接关系的矩阵称邻接矩阵

#### 图非网时

用 boolean 矩阵表示

设图有  $n \geq 1$  个顶点，则图 G 的邻接矩阵定义为 n 阶方阵 M，M 满足下列性质：

$$m_{ij} = \begin{cases} 1, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E(G) \\ 0, & \text{反之} \end{cases}$$

对于无向图，它的邻接矩阵一定是一个对称矩阵，且矩阵第 i 行之和是顶点  $v_i$  的度

对于有向图，它的邻接矩阵一定是一个非对称矩阵，矩阵第 i 行之和是顶点  $v_i$  的出度，矩阵第 i 列之和是顶点  $v_i$  的入度。

#### 网的邻接矩阵

把原来的存 1 改成存权值。

设图有  $n \geq 1$  个顶点，则图 G 的邻接矩阵定义为 n 阶方阵 M，M 满足下列性质：

$$m_{ij} = \begin{cases} w_{ij}, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E(G) \\ 0 \text{ 或 } \infty, & \text{反之} \end{cases}$$

在计算机中，可以取一个相对于问题而言相对大的数即可代表。

### 8.2.2 邻接表

对图的每个顶点建立一个链表，也称邻接链表表示。

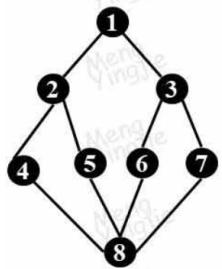
有  $n$  个结点时有  $n$  个链表，第  $i$  个链表中的结点：

- 在无向图中：是与  $v_i$  邻接的所有结点的收集。
- 在有向图中：是以  $v_i$  为始点的所有终点的收集。

## 无向图

### 二. 邻接表(Adjacency List)

**举例:** 无向图



第*i*个链表中的结点个数:  
总表结点数目:



开头的结点是线性结构，可以用向量或者链表存，此处用向量存开头的节点。

总链表结点数目:  $2e$  (每条边出现了两次)

加上表头，存储空间需  $n+2e$

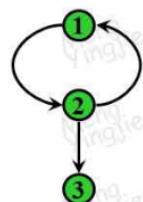
第*i*个链表中的结点个数:  $v_i$  的度。

无向图的邻接表，链表中的顺序可以随意变换，不分先后（即同一个图也可以有不同的邻接表）

结点间是否有边判别比邻接矩阵困难。

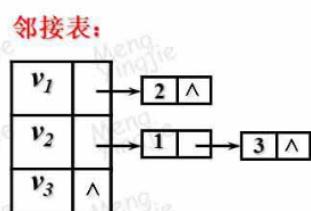
## 有向图

**举例:** 有向图



第*i*个链表中的结点个数  
总表结点数目:

**存储示意图:**



对于网的邻接表，表结点可以再增加一个数据项即可。

表节点数目:  $e$

总空间:  $n+e$

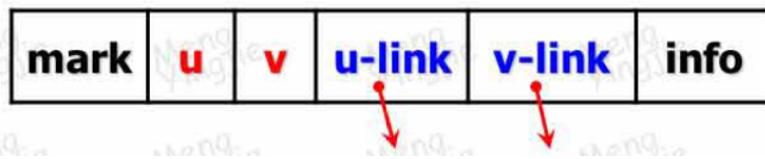
第  $i$  个链表中的结点个数:  $v_i$  的出度。

逆邻接表: 以  $v_i$  为终点的所有始点的收集。可用于获取邻接过来的结点。

### 8.2.3 邻接多重表

邻接多重表以边为存储单位。

边  $(u, v)$  结点结构:



- mark: 标志域, 标记该边是否被处理, 也可存放权值等信息
- u, v: 关联边的两个顶点域
- ulink: 下一条与依附于 u 的边的地址 (指针)
- vlink: 下一条与依附于 v 的边的地址 (指针)
- info: 其他信息

## 8.3 图的遍历

### 8.3.1 概述

#### 定义

给出图  $G$  和其中的任意一个顶点  $v_0$  (人为给出), 从  $v_0$  出发系统地访问  $G$  中所有的顶点, 且每个顶点仅被访问一次, 这一过程称为图的遍历 (graph traversal), 或遍历图。

#### 辅助结构

须设立辅助结构, 可用数组  $v_{isited}[1..n]$  表征

$$\text{visited}[v] = \begin{cases} 1, & v \text{已经被访问} \\ 0, & v \text{未被访问} \end{cases}$$

#### 搜索策略

- 深度优先搜索 (DFS, depth-first search)

搜索中，结点扩展的次序向某一个分支纵深推进，到底后再回溯。

- 广度优先搜索 (BFS, breadth-first search)

搜索中，对所在层次的所有结点逐个依次进行扩展后，再推进到下一个层次进行扩展。

### 8.3.2 图的深度优先遍历

#### 图的深度优先遍历的基本思想

1. 访问出发节点  $v_0$ ，标记  $v_0$  为已访问
2. 选择一个  $v_0$  邻接到的且未被访问过的结点  $u$
3. 从  $u$  开始进行深搜（递归开始）

#### 基于邻接表

面授部分讲解

### 8.3.3 图的广度优先遍历

#### 图的广度优先遍历的基本思想

1. 访问出发点  $v_0$
2. 访问  $v_0$  邻接到的所有未被访问过的节点  $v_1, v_2, \dots, v_t$
3. 再依次访问  $v_1, v_2, \dots, v_t$  邻接到的所有未被访问的结点
4. 如此进行下去，直到无法找到未被访问的结点时，则本次搜索就算结束

## 8.4 连通性及最小生成树

### 8.4.1 连通分量和生成树

#### 求图的连通分量

思想：将图的遍历中搜索算法 DFS(或 BFS) 中的访问动作具体化 (即输出顶点及与之关联的边)，每启动一次 DFS(或 BFS) 就可以得到 1 个连通分量。

#### 求生成树

通过图的遍历可获取其生成子图。

对于连通图来说该生成子图，实际上是一棵树结构 (恰好有  $n-1$  条边，它们把  $n$  个事物关联起来，可称其是原图的极小连通子图)

以下三种情况：

- 图是连通的

- 图是强连通的
- 出发点  $v_0$  为图根

在遍历过程中，访问的节点和经过的边，所构成的生成子图，恰好可以构成一个树形结构，称为生成树。对于有  $n$  个顶点的连通图，其生成树具有  $n-1$  条边，即生成树是图的最小结构。

对于不连通的无向图和不是强连通的有向图，从任一结点出发只能得到生成森林。

生成树因为出发点不同，深、广不同而不唯一。

基于遍历方法我们可以构造极小连通子图（一般连通图一定是无向图）（即节点不变，关联关系最小）。

#### 8.4.2 最小生成树

最小生成树是边的权值之和最小的网络的生成树

##### 普里姆 (Prim) 算法

基本思想：设  $V=1,2,\dots,n$  是图  $G$  的顶点集合，PRIM 算法由一个初值为  $v_0$  的集合  $U$  开始，它每次生成一条边，逐渐长成一棵具有最小代价的生成树。

算法在每一步中都找出一个最小权的边  $(u,v)$ ，其中  $u \in U$ ,  $v \in V-U$ . ( $U,V$  是点集， $U$  是已经找到的点的集合， $V$  是所有点)

即从任意一个点出发找到已找到的点集到未找到的点集中权值最小的一条路径不断连接即可

$v_0$ : 种子节点，可任选

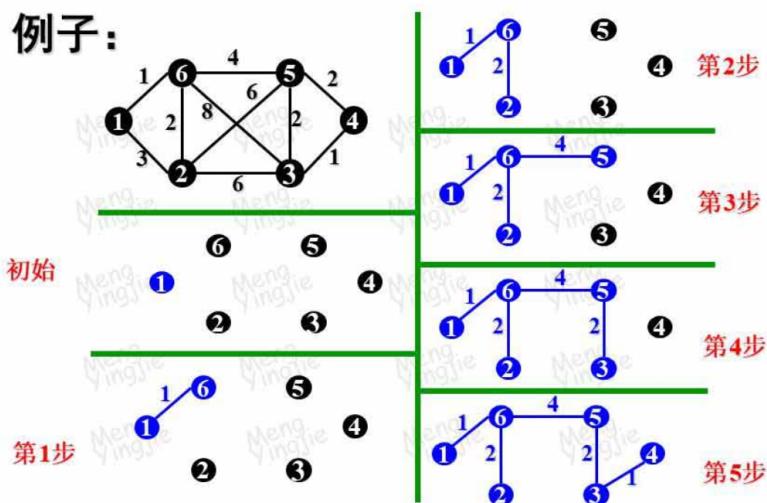


图 8.2 普里姆 (Prim) 算法例子

### 克鲁斯卡尔 (Kruskal) 算法

基本思想：对于图  $G=(V,E)$ ，该算法初始化从  $T=(V, \emptyset)$  开始，此时生成树是由  $n$  个连通分量（即  $n$  个孤立顶点组成）组成的一个制图——目标使得  $n$  个连通分量逐渐成为一个连通分量。具体做法：

1. 初始化从  $T=(V, \emptyset)$  开始，即  $E=\emptyset$
2. 按照权值递增的顺序逐个考虑  $E$  中的每条边：
  - (a) 若该边连通了在两个不同连通分量中的顶点，则将该边填加到  $T$  中
  - (b) 重复 (1)，一旦  $T$  中包含了  $n-1$  条边，则终止运算

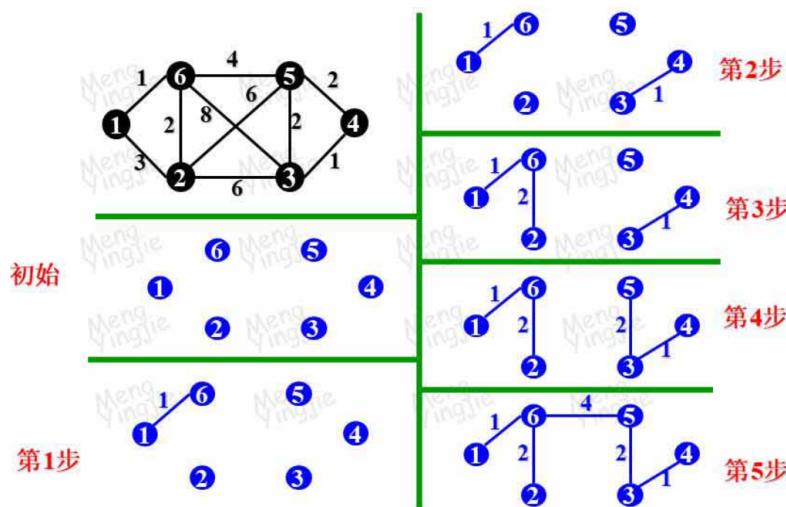


图 8.3 克鲁斯卡尔 (Kruskal) 例子

## 8.5 有向无环图及应用

一个无环的有向图，我们称为有向无环图 (DAG, directed acyclic graph)。这一节只讨论 DAG。是介于树形结构和有向图之间的一种模型，在实际工程应用中价值比较高。

### 8.5.1 拓扑排序

#### 相关术语

- AOV-网

若有向图  $G$  中，顶点表示活动或任务，有向边表示活动或任务之间的优先关系，则此有向图称为顶点表示活动的网络。(AOV-网，Activity On Vertex Network)

一个 AOV-网要能够表达一个可执行的工程则其优先关系应当是非自反的。

若存在回路，则说明某项活动的能否进行是要以自身任务的完成作为先决条件。对一个程序来说相当于出现了死循环。因此给定一个 AOV-网，当要检测该网所表示的工程是否可以实现时，就可以通过检查其是否有回路来完成。一种有效的方法就是拓扑排序（寻找图的偏序序列的过程）。

- 拓扑序列

对于有向图  $G=(V,E)$ ,  $V$  中的顶点的线性序列  $(v_{i1}, v_{i2}, \dots, v_{in})$ ，称作一个拓扑序列，若此结点序列满足如下条件：在  $G$  中从顶点  $u$  到顶点  $v$  有一条路径，则在序列中  $u$  必在  $v$  之前。

拓扑序列不唯一

任何无环的有向图，其顶点都可以排在一个拓扑序列中

- 拓扑排序

寻找图的拓扑序列的过程。图中的拓扑序列往往不止一个。拓扑排序只需要找到一个。

### 拓扑排序基本思想

1. 从图中选择一个入度为零的顶点
2. 输出该顶点，从图中删除此顶点及其所有的出边

反复执行以上两步，直到所有顶点都输出，此时拓扑排序完成；或者直到剩下的图中再无入度为零的顶点，此时说明图  $G$  是有环的图，拓扑排序无法完成。

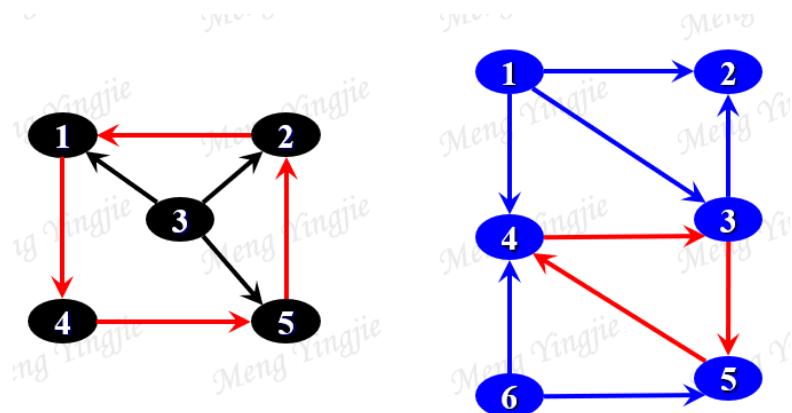


图 8.4 死锁实例

### 拓扑排序实现——基于邻接矩阵

基于基本思想中的关键两步，进行具体细化：

1. 选择入度为零的结点  $u$ ——选择全零的列  $u$ 。（行发出，列接收）
2. 输出顶点，删除出边

## (a) 顶点的输出

策略：拓扑序列可通过辅助数组  $S[1..n]$  保存，具体取值：

$$S[i] = \begin{cases} 0, & \text{该节点未获输出} \\ m, & m \text{ 为第 } v_i \text{ 个结点在拓扑序列中的输出次序} \end{cases}$$

(b) 删除出边，删除  $u$  所有出边

删除  $u$  所有出边即讲邻接矩阵中第  $u$  行置零。

算法步骤：

1. 辅助数组  $S$  置零，输出次序计数器置初值（即：设定顶点的输出的顺序编号的起步值）
2. 寻找没有输出编号的全零的列  $u$ ，如果没有，则算法终止。此时  $S$  中所有元素都有输出编号，则拓扑排序完成；否则有环。
3. 将输出次序号赋给  $S[u]$ 。
4. 把第  $u$  行置成全零。
5. 输出次序号加 1，回到 2。

算法复杂度为  $O(n^3)$

## 拓扑排序实现——基于邻接表

## • 预处理工作

## – 存储结构的优化

要获取入度，需有逆邻接表。要删除所有出边，需有邻接表。两个需求无法同时满足。于是做以下处理：选择邻接表，在图建立的时候加一个入度项，计算每个结点的入度。

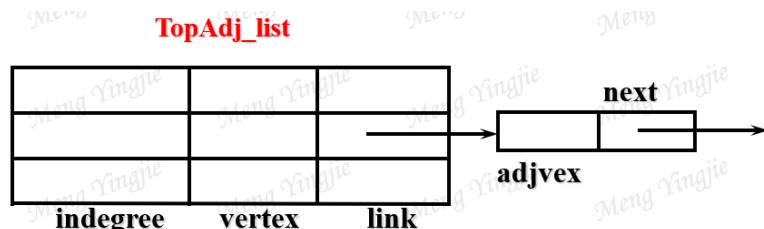


图 8.5 邻接表结构设计

## – 设置一个集合存放参与运算的结点

需要一个结构 (S) 保存其它入度为零的顶点，以备选用。S 可以采用栈。

- 运算的步骤

1. 搜索邻接表中入度为零的顶点，并令其进栈
2. 当栈非空时，进行拓扑排序：
  - (a) 退栈，输出栈顶元素 u
  - (b) (循环) 在邻接表中扫描 u 的直接后继顶点 k
    - 将 k 入度减一
    - 若此时 k 的入度为零，则 k 入栈
3. 若栈空，输出元素不足 n 则有环，否则拓扑排序完成

- 栈结构的优化设计

为了保存待输出的入度为零的结点 (可保存在栈 S 中)，但这需要额外的空间。

邻接表中的入度域是要参与运算的 (减法)，因此在拓扑排序后，该域将失去实际意义；另外入度为 0 的顶点的入度域是不会参与运算的，因此，S 可借用这些单元作为栈的存储空间；这样可以将入度为零的顶点的入度域组织成一个静态的链栈。此时 indegree 域相当于 next 域的作用。

### 8.5.2 关键路径

#### AOE-网及关注的问题

与 AOV-网对应的是 AOE 网。AOE-网在企业管理、工程计划等当中有广泛的应用。

若在带权有向图中顶点表示事件，有向边表示活动，权表示活动持续的时间，则此有向图称为边表示活动的网络 (AOE-网，Activity On Edge Network)。

表示实际工程 (或计划) 的 AOE-网，应该是无环的，且存在唯一入度为零的起始顶点 (始点)，以及唯一的出度为零的完成顶点 (终点)。

关注的点：

- 利用事件 AOE-网可以进行工程安排估算，研究完成整个工程至少需要多少时间
- 为缩短整个工程的完成时间，应该加快哪些活动的速度

在 AOE-网要解决这些问题，可以采用多种技术：

- PERT(Program Evaluation and Review Technique)(程序评价和审定技术)
- CPM(Critical Path Method)(关键路径法)

- RAMPS(Resources Allocation and Multi-Project Scheduling )(资源分配和多项目调度)

## 8.6 最短路径

### 8.6.1 单源最短路径

单源最短路径是指从一个顶点到其它各顶点之间的最短路径 (single-source shortest path)。

迪杰斯特拉 (E.W.Dijkstra) 于 1959 年提出了一个寻找单源最短路径的方法。

其基本思想是，设置一个顶点集合  $S$ ，并不断地作贪心选择来扩充这个集合。该算法属于算法设计方法中的贪心算法 (greedy selector) 类——总是作出当前看来最好的选择，通过获取局部最优，最终达到获取整体最优)。

### 8.6.2 每对顶点的最短路径 (all-pairs shortest path)

## 第九章 排序

### 9.1 基本概念

#### 9.1.1 概论

排序是数据结构运算层面的问题。

排序 (sorting) 是计算机程序设计中的一种重要的运算，其功能是将一个数据元素的任意序列，按照要求重新排列成按一定规则有序的序列。

排序 (又称分类) 通常可以理解为：根据与项目中所包含的关键字或信息项的有关规则，对信息项目加以排列整理的动作过程。

#### 9.1.2 相关概念

##### 关键字 (key)

如果数据元素 (或结点、记录) 中的某个数据项的值可以用它标识一个数据元素，则将该数据项称为关键字。

关键字这一项往往不反应事物的本质。

##### 主、次关键字

- 主关键字

可以唯一地标识一个记录的关键字称为主关键字 (Major Key、Primary Key)，实际应用中关键字不加声明时都指的是主关键字。Major key: 在一个记录中的最主要的关键字；

Primary key: 在文件组织中，进行大量访问时所用关键字。

- 次关键字

次关键字 (辅助关键字、Secondary Key): 可以识别若干记录的关键字称次关键字。

##### 排序

设含有  $n$  个记录的集合为  $R=r_1, r_2, \dots, r_n$ , 其对应的关键字集合为  $K=k_1, k_2, \dots, k_n$ , 给定关系  $\alpha$ , 按照关系  $\alpha$  针对关键字集合  $K$  对  $R$  进行运算, 使得  $R$  有如下序列:  $(r_{\alpha 1}, r_{\alpha 2}, \dots, r_{\alpha n})$ , 我们将这个操作过程称为排序。

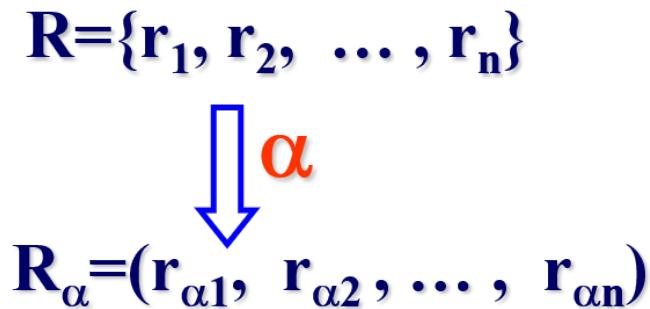


图 9.1 排序的本质

本质是在关系  $\alpha$  下寻找  $R$  的一种排列的过程。

### 排序的稳定性

在排序关系下，假设排序前  $r_i$  在  $r_j$  之前，排序之后领先关系不变，则称此排序过程和排序方法是稳定的，否则是不稳定的。

排序前	排序后	排序后
$A_1:85$	$A_2:90$	$A_2:90$
$A_2:90$	$A_1:85$	$A_3:85$
$A_3:85$	$A_3:85$	$A_1:85$
$A_4:78$	$A_4:78$	$A_4:78$
稳定的		

图 9.2 排序的稳定性例子

### 排序的分类

排序一般涉及大量数据，而大规模数据一般都存放于外部存储器上，并以文件的形式出现。因此，依据排序过程中数据所处位置，可以将排序分为两大类：

- 内(部)排序：

将整个待排序的文件装入内存，并在其中进行排序，这种排序过程称为内(部)排序 (internal sorting)。

- 外(部)排序：

由于数据规模过大，排序过程不能在内存中一次完成，需要不断进行内外存数据交换才能完成排序，这样的排序过程称外(部)排序 (external sorting)。

## 排序的应用

- 作为检索的辅助手段
- 作为数据项匹配的手段

## 影响排序的因素

### 9.1.3 存储结构设计

排序运算可以遵循前面运用的一些存储结构，但有时为了保证高效的排序，可能还需要设计一些特殊的存储方式。

排序常用的存储结构有以下几种：

#### 向量

待排序的初始文件各记录依其自然顺序存放在连续的一块地址空间中。

#### 链表结构

记录以结点形式按记录原始次序链接起来。

#### 地址向量结构

将要排序文件的各个记录存储到内存的各个块中，这些块的地址一般是不连续的。按各记录的原始次序，将这些的块的首地址依次存入内存的一块连续单元中，由各块的首地址组成了一个向量——地址向量。

这样可实现局部连续，整体不连续的组织模式，这种组织方式具有较高的实用性。

就是原来的索引结构

## 9.2 插入排序

### 9.2.1 插入类排序基本方法

**基本思想：**

1. 每次只考虑一个待排记录  $r$
2. 将  $r$  按照排序关系插入到一个已经有序的文件适当位置
3. 重复上述过程直到全部记录插完为止

### 9.2.2 直接插入排序 (straight insertion sort)

1. 将要排序的源文件  $F=R_1, R_2, \dots, R_n$ , 视为两部分：

$$F' = R_1; F'' = R_2, \dots, R_n$$

2. 对  $F'$  和  $F''$  重复如下工作：

- (a) 从  $F''$  中取出一个记录  $R_i$ , 并将  $R_i$  从  $F''$  中删除
- (b) 将  $R_i$  插入到  $F'$  并使  $F'$  线性有序

第  $i$  个记录的插入过程：

1. 进行插入准备：先取出第  $i$  个元素放入  $x$  临时存储 ( $x \leftarrow R[i]; j \leftarrow i-1$ )

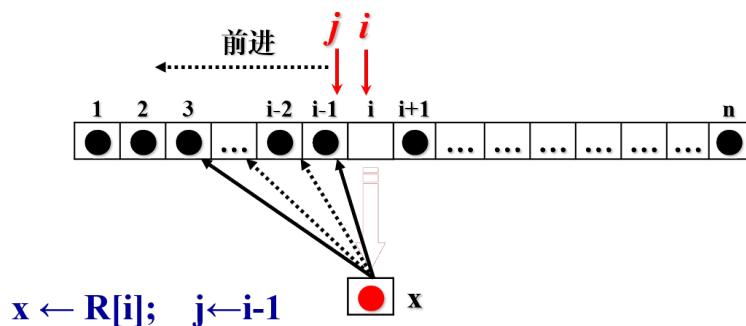


图 9.3 将第  $i$  个元素放入  $x$  临时存储

2. 寻找插入位置：向前进行比较，条件： $(x.key < R[j].key) \text{ and } (1 \leq j)$
3. 插入元素归位： $R[j+1] \leftarrow x$

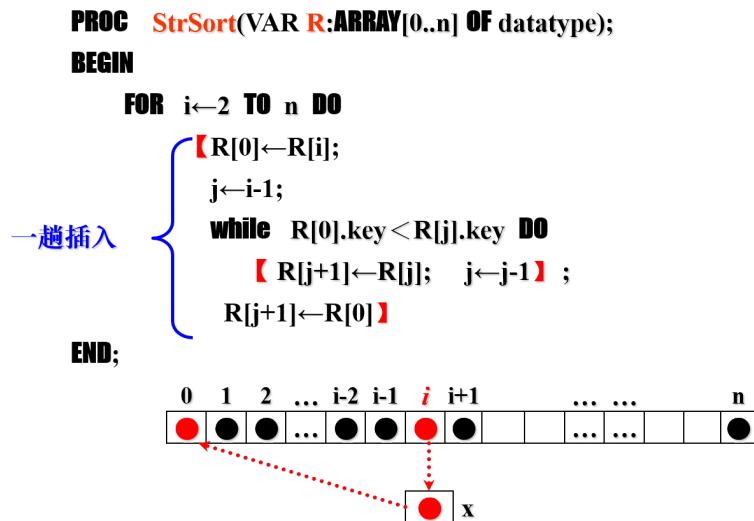


图 9.4 直接插入算法过程

### 9.2.3 二分插入排序 (dichotomising insertion sort)

也称折半插入排序。与直接插入排序的区别：在插入第  $i$  个时搜索采用二分策略。

仅对比较次数有改善，移动次数无影响。算法过程如下：

```

PROC DichSort(VAR R:ARRAY[1..n] OF datatype);
BEGIN
  FOR i←2 TO n DO
    {
      x←R[i]; l←1; h←i-1;
      while l≤h DO
        m←(l+h) DIV 2;
        IF x.key<R[m].key THEN h←m-1
        ELSE l←m+1 END IF;
        FOR j←i-1 DOWNTO l DO R[j+1]←R[j];
        R[l]←x
    END;
END;

```

比较时间  $O(n \log_2 n)$   
移动时间  $O(n^2)$

图 9.5 二分插入算法过程

#### 9.2.4 2-路插入排序

在二分插入排序的基础上再改进。

目的：减少排序过程中记录的移动次数。

借助辅助空间 D[1..n]。

#### 9.2.5 表插入排序 (list inserting sort)

采用链式结构来组织待排序的数据元素。

是一种较为古典的算法，仅适用于链结构，下完成分类。

其独到之处是利用链结构的特点，通过变更记录指针来调整记录的逻辑顺序，从而在不发生记录迁移的情况下完成分类。

#### 9.2.6 希尔排序 (Shell sort)

又称缩小增量法 (diminishing increment sort)，是一种快速的排序方法，1959 年由 D.L.Shell 提出。

##### 基本思想

把对源文件 F 的排序分成多步来完成，算法在每步中取一个步长 d，将 F 逻辑上看成 d 个文件。

1. 按照插入排序的办法把这 d 个文件分别排序；
2. 然后缩小 d
3. 重复 1-2，直到 d=1 的一次排序为止

### 9.3 交换排序

#### 9.3.1 交换类排序基本方法

**基本思想：**

每次考虑两个待排序的记录。

依据排序关系两两比较待排序记录，并交换不满足顺序要求的那些偶对，直到全部满足为止。

#### 9.3.2 起泡排序 (bubble sort)

**基本思想：**

先比较 R1 与 R2, 如果 R1 大, 则交换 R1 和 R2; 然后对 R2 和 R3 做同样的处理, 重复此过程直到处理完 Rn-1 和 Rn 的比较和交换。

这样从 (R1, R2), (R2, R3), 直到 (Rn-1, Rn) 的 n-1 次比较和交换过程我们称为一趟起泡。

一趟起泡的明显结果是将最大的传到了最后 (最终位置)。

显然, 第二趟起泡只需进行 (R1, R2), (R2, R3), …, (Rn-2, Rn-1) 的比较和交换。

这样最多做 n-1 次起泡即可得到最终有序序列。

若一趟起泡中没有发现元素交换, 则已经有序, 可以直接终止交换。

**分析**

- 算法是稳定的;
- 算法时间复杂性分析:

$$\text{比较次数: } \left\{ \begin{array}{l} C_{\min} = n-1 \\ C_{\max} = \sum_{i=1}^{n-1} (n-i) \approx n^2/2 \end{array} \right.$$

$$\text{移动次数: } \left\{ \begin{array}{l} M_{\min} = 0 \\ M_{\max} = \sum_{i=1}^{n-1} 3i \approx 3n^2/2 \end{array} \right.$$

- 算法空间复杂性分析: O(1)

### 9.3.3 快速排序 (quick sort)

#### 概述

也称分区交换排序，或 Hoare 排序。1962 年首先由霍尔 (C.A.R.Hoare) 提出。是至今为止内部 (比较) 排序中较快的一种。它有广泛的应用，典型的应用是 UNIX 系统调用库函数例程中的 qsort 函数。但快速排序往往由于最差时间代价的性能而在某些应用中无法采用。越接近有序性能越差。主要特点是采取了分治的思想。

#### 快排基本思想：

(以下比较指关键的比较，关键字指排序关键字)

1. 在待排序的  $n$  个记录中任取一个记录  $r$  (例如就取第 1 个)，作为轴心元素
2. 以  $r$  为标准将所有记录分为两组。第 1 组中各记录的关键字都小于  $r$  的关键字；第 2 组中各记录的关键字都大于  $r$  的关键字
3. 并把  $r$  排在这两组中间 (最终位置)，这一过程称为一趟快排
4. 然后对这两组分别重复上述方法，直到所有记录都排在应有位置上

#### 一趟快排描述

初始准备：

- 建一个额外空间  $x$  存放取出来的任意元素
- 设置两个指针  $i, j$

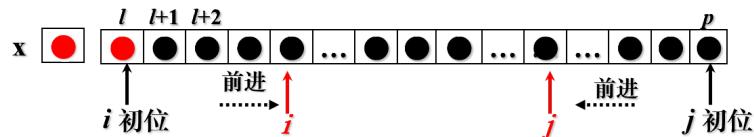


图 9.6 初始准备

处理：

1. 当 ( $i < j$ ) 且  $R[j].key > x.key$  时， $j$  指针前进
2. 如果  $i < j$ ，则：
  - (a)  $R[j] \Rightarrow R[i]; i+1 \Rightarrow i$
  - (b) 当 ( $i < j$ ) 且  $R[i].key < x.key$  时， $i$  指针前进
  - (c) 如果  $i < j$ ，则： $R[i] \Rightarrow R[j]; j-1 \Rightarrow j$
3. 如果  $i=j$ ，则本趟结束，否则转

4. 以上结束后:  $x \Rightarrow R[i]$

### 快排算法过程

## 9.4 选择排序

### 9.4.1 选择类排序基本方法

每次考虑一个数据元素的最终位置。

每步从待排序的记录中挑选一个当前最小(或最大)的结点，将其放在应有位置，直到全部记录排完为止。

### 9.4.2 直接选择排序 (straight selection sort)

首先在所有记录中选出最小的记录；

把它与第一记录交换；

然后在剩下的记录中再送出(全局)次最小的记录与第二个记录进行交换；

依次类推，直到全部记录排完为止。

### 9.4.3 树形选择排序 (tree selection sort)

### 9.4.4 堆排序 (heap sort)

## 9.5 合并排序

### 9.5.1 合并类排序基本方法

合并排序(或归并排序, merge sort)要求待排序文件已经部分排序。首先把一个集合中的数据元素分成若干个子集，对每个子集中的元素进行排序，再将所得到的有序子集进行合并。

### 9.5.2 两组合并

### 9.5.3 二路归并排序

如果文件中有  $n$  个记录，可将文件看成  $n$  个有序的子文件，这样就出现文件“部分有序”的状态；因而可以使用合并排序。

不过合并  $n$  个有序段，其操作较为复杂；

通常可以先将每两个子文件进行合并，得到  $n/2$  个部分有序的较大子文件，每个子文件含有两个记录；

再将这些子文件合并，如此反复，直到最后合并成一个文件时，排序就完成了；

## 9.6 枚举排序

### 9.6.1 计数类排序基本方法

利用统计的方法，即统计小于（或大于）关键字 K 的个数。

由此可见，计数排序（或枚举排序,merge sort）的基础在于比较关键字和计数。

为此，每个记录需带一个计数器（count，存放小于该记录的排序关键字的记录个数）。

### 9.6.2 计数类排序的过程

## 9.7 分配排序

### 9.7.1 分配排序概述

## 第十章 数据检索

### 10.1 基本概念

#### 10.1.1 概论

本章集中讨论非数值程序设计中的另一个重要的技术问题——检索 (search, 或称为查找 seek).

#### 10.1.2 相关概念

##### 检索

在给定数据结构中查找满足某种条件的数据元素 (或结点、记录) 的过程。

##### 检索的分类

- 基于关键字的检索

在给定的结构中找出关键字等于指定值的结点。检索成功时，往往只得到一个结点即可

- 基于属性的检索

在给定结构中找出某属性值等于指定值的结点。检索成功时，检索结果往往是一批结点

##### 检索方法的分类

根据检索对象的组织关系和检索对象的存储组织模式，检索算法有三类：

- 顺序表和线性表方法

- 直接访问法 (散列方法);

- 树索引方法;

##### 检索算法的特性

- 内外有别

分内检索和外检索。内检索是内存能够容纳全部记录的情形。

- 静态动态

静态检索时，表的内容不变 (即一个单纯的查找过程)；动态检索时，表中的内容不断地在变动 (即表有频繁地插入/删除记录的操作)。

- 原字变字 s 原字系指用原来的关键字；所谓变字是指使用经过变换过的关键字。
- 数字文字

指比较时用不用数字的性质。用数字的性质就是象排序算法中那样作各位数的分布计数，而不是直接对关键字进行比较，例如字符树就是使用数字性质。

### 检索算法效率的度量

平均检索长度 (ASL, Average Search Length): 检索过程中对关键字 (或属性) 要执行的平均运算次数。

$$ASL = \sum_{i=1}^n (C_i \times P_i), \text{ 其中, } C_i \text{ 查找第 } i \text{ 个的比较次数, } \\ P_i \text{ 查找第 } i \text{ 个的概率}$$

这里的运算在大多数情况下是关键字的比较运算。Pi 一般认为是等概率的 (即  $P_i=1/n$ )

### 相关概念

## 10.2 线性表的检索

### 10.2.1 概论

### 10.2.2 顺序检索

#### 检索方法

用给定的关键字值与线性表中各结点的相应关键字值进行逐一比较。

找到相等的则检索成功，否则检索失败。

这种方法对顺序分配或链接分配都是适应的。该方法对于待检索文件的结点无排序要求。

#### 向量存储时的算法

##### 简单分析

设每个记录检索概率相等。

检索成功的比较次数:  $ASL = \sum_{i=1}^n \frac{i}{n} = \frac{n+1}{2}$

检索不成功的比较次数:  $n+1$ .

时间复杂度: $O(n)$

空间复杂度: $O(1)$

算法优点：简单易行。

缺点：检索时间长，检索长度与表中结点数成正比。

### 10.2.3 二分检索

#### 检索方法

##### 二分检索的基本作法

此方法是将要检索的对象分成两部分，舍弃不包含所需要项目的那一部分，对剩下的部分再用相同的方法进行划分，直到找到所需要的项目或划分部分为空为止。也称对分检索、折半检索。

#### 算法过程

#### 简单分析

设每个记录检索概率相等。

平均检索长度:  $ASL \frac{n+1}{n} \log_2(n + 1) - 1$

n 较大时,  $ASL = \log_2 n$

为换取快速检索所付出的代价是要将线性表排序；适应于一旦建立起来就很少改动而又需要经常检索的线性表。

#### 有序表的其它检索方法

- 斐波那契 (Fibonacci) 检索
- 插值检索

### 10.2.4 分块检索

既有较快的速度，又能够动态变化——比较实用

#### 分块的组织

分块检索要求把线性表分成若干块，在每一块中记录的存放是任意的，但是块与块之间必须有序。

## 10.3 树形结构的检索

### 10.3.1 概论

树形结构的一个重要应用就是用来组织目录和符号表。

树形结构的检索就是针对树目录和树表所组织的检索。

以树形结构组织的目录我们称为树目录；以树形结构组织的符号表我们称为树表。

### 10.3.2 二叉排序树

二叉检索树的检索效率取决于该二叉树的深度，但该二叉树是动态生成的，由于输入的结点次序的不同可能导致出现不同的二叉树，也可能出现歪树，使其检索性能退化到与顺序检索等同的情况。优势：

- 性能好
- 保证动态有序性

### 10.3.3 AVL 树

这种二叉树它的子树的深度是平衡的，称为平衡二叉树（即 AVL-树，或称为均高二叉树）。

按照这种平衡结构构造的二叉树称为最佳二叉排序树或最优二叉排序树。

#### 概念

- 结点的平衡因子：  
结点的左子树高度减去右子树的高度的差值。
- AVL-树：
  - 一棵空二叉树是 AVL-树
  - 若  $T$  是一棵非空二叉树，其任何结点的左、右子树的高度相差不超过 1，则  $T$  是 AVL-树

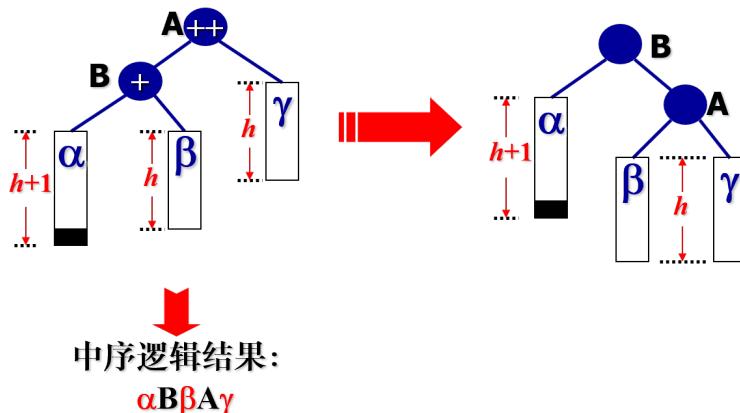
#### AVL-树平衡的保持

在平衡的二叉树中若进行插入或删除将可能导致不平衡情况的出现，这时就要运用一定的规则进行调整。

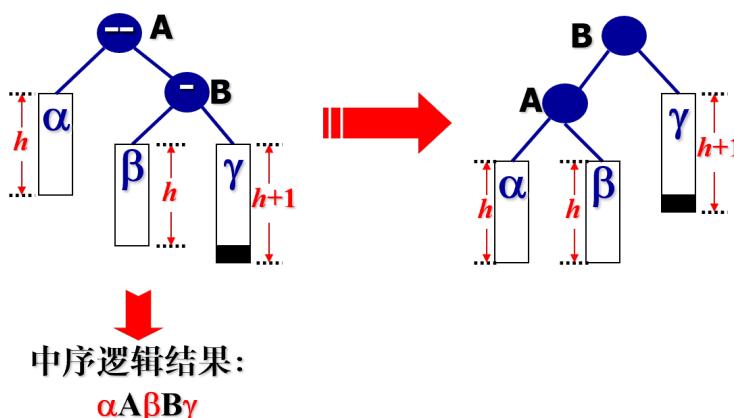
- 平衡规则  
选择离插入（或删除）结点最近的不平衡结点（其平衡因子为  $\pm 2$ ）开始调整。
- 平衡类型
  - LL 型：新结点插在 A 的左子树的左子树中导致不平衡
  - RR 型：新结点插在 A 的右子树的右子树中导致不平衡
  - LR 型：新结点插在 A 的左子树的右子树中导致不平衡
  - RL 型：新结点插在 A 的右子树的左子树中导致不平衡

### 调整过程

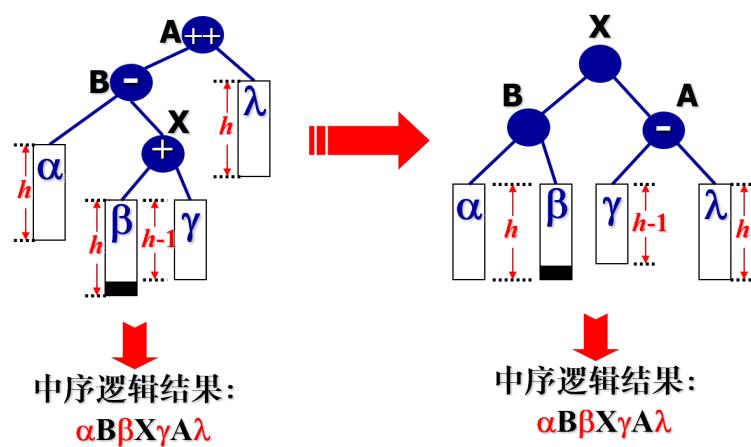
- LL 型:



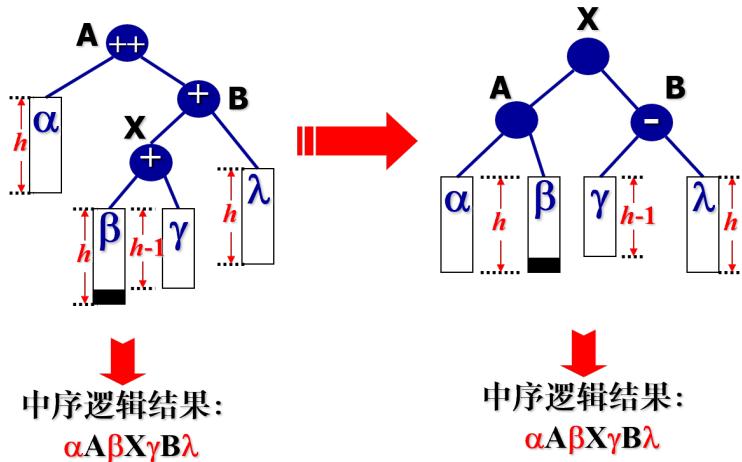
- RR 型:



- LR 型:



- RL 型:



总结得:



## 10.4 散列表的检索

### 10.4.1 概论

既是存储方法，也是检索方法。

#### 基本思想

将关键字看成一个变量，通过一定的函数关系，将函数值解释为存储地址，将结点存入这样计算得到的地址单元中，检索的过程是存储过程的逆过程。

也称关键字——地址转换法，或关键转换法。

在散列法中把地址映射函数称为散列函数 (hash function)。

把用散列法组织存储的线性表称为散列表 (hash table)。

### 10.4.2 散列函数

#### 直接定址

取关键字的某个线性函数值。 $H(k)=a*k+b$ ,  $a,b$  为常数。也称自身函数。

#### 除余法

选择一个适当的正整数  $p$ , 用  $p$ 去除关键字, 取其余数作为地址, 即 modulo- $P$  residue , 可形式化表示为:

$$H(k)=k \bmod P$$

#### 数字分析法

对各个关键字内部代码的各位进行分析, 抽取分布比较均匀的若干位作为地址, 抽取的位数取决于地址码的位数。

#### 分划法

也称折叠法。若关键字很长, 且可变的; 或者关键字由多个域组成, 此时散列技术可将关键字的内部代码分割成多个部份, 并将这些部份按某种规律结合起来。

#### 平方取中法

也称中平法。先计算出关键字的平方值, 再抽取它的中间若干几位作为散列地址。

#### 基数转换法

关键字的符号组成一般是建立在一定基数制上的, 我们可以将该关键字看成是另一个基数制上的表示, 再将其还原为原来基数制上的表示后, 抽取其若干位作为散列地址。

#### 随机数法

### 10.4.3 碰撞的产生及处理

#### 碰撞及其产生

依据散列函数  $H$  计算出地址, 若发现此地址已经被别的结点占用, 即就是说有两个不同的关键映射到了同一地址空间, 我们把这种现象称为碰撞 (或冲突,collision)。

### 碰撞的处理

当碰撞发生后处理碰撞的方法基本上有两类：拉链法和开地址法，也有分为开散列 (open hashing, 基本区外) 和闭散列 (closed hashing, 基本区内) 的。

- 拉链法

当碰撞发生时，就拉出一条链，建立一个链接方式的同义词子表。

- 开放地址法

当碰撞发生时，用某种方法形成一个探测的序列，沿着这个序列一个个单元地查询，直到找到这个关键字或者找到一个开放的地址 (open addressing, 即没有进行存储的空单元)。

## 10.5 基于属性的检索

## 参考文献

- [1] 严蔚敏, 吴伟民. 数据结构第二版 [J]. 清华大学出版社 2001—1. 1992..

## 致 谢

感谢蒙老师的教导；感谢物理院余航学长开源的 LaTeX 模板；感谢我电脑没有在中途死机；感谢我的朋友们对我的支持。

## 论文（设计）成绩表

建议成绩_____		
指导教师（签字）_____		
答辩小组意见		
答辩委员会负责人（签字）_____		
成绩_____	学院（盖章）_____	年   月   日